PERFORMANCE UNDERSTANDING AND TUNING OF ITERATIVE COMPUTATION USING PROFILING TECHNIQUES

A Thesis
Presented to
The Academic Faculty

by

Sarang Anil Ozarde

In Partial Fulfillment
of the Requirements for the Degree
MS in Computer Science in the
College of Computing

Georgia Institute of Technology
August 2010

PERFORMANCE UNDERSTANDING AND TUNING OF ITERATIVE COMPUTATION USING PROFILING TECHNIQUES

Approved by:

Dr. Santosh Pande, Advisor
College of Computing
Georgia Institute of Technology

Dr. Sudhakar Yalamanchilli
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Nathan Clark
College of Computing
Georgia Institute of Technology

Date Approved: 12th May 2010

To my parents,
For their invaluable and never ending support.

ACKNOWLEDGEMENTS

I would like to heartily thank my advisor Dr. Santosh Pande for providing me his valuable guidance and constant support throughout my Masters. I would like to thank my colleague Romain Cledat who helped me in building the infrastructure for validating my hypothesis. I would like to thank my committee members Dr. Sudhakar Yalamanchilli and Dr. Nathan Clark for their valuable time and feedback.

This thesis would not have been possible without encouragement and support from my family and friends and I owe my deepest gratitude to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Most applications spend a significant amount of time in the iterative parts of a computation. They typically iterate over the same set of operations with different values. These values either depend on inputs or values calculated in previous iterations. While loops capture some iterative behavior, in many cases such a behavior is spread over whole program sometimes through recursion. Understanding iterative behavior of the computation can be very useful to fine-tune it. In this thesis, we present a profiling based framework to understand and improve performance of iterative computation. We capture the state of iterations in two aspects 1) Algorithmic State 2) Program State. We demonstrate the applicability of our framework for capturing algorithmic state by applying it to the SAT Solvers and program state by applying it to a variety of benchmarks exhibiting completely parallelizable loops. Further, we show that such a performance characterization can be successfully used to improve the performance of the underlying application.

Many high performance combinatorial optimization applications involve SAT solving. A variety of SAT solvers have been developed that employ different data structures and different propagation methods for converging on a fixed point for generating a satisfiable solution. The performance debugging and tuning of SAT solvers to a given domain is an important problem encountered in practice. Unfortunately not much work has been done to quantify the iterative efficiency of SAT solvers. In this work, we develop quantifiable measures for calculating convergence efficiency of SAT solvers. Here, we capture the Algorithmic state of the application by tracking the

assignment of variables for each iteration. A compact representation of profile data is developed to track the rate of progress and convergence. The novelty of this approach is that it is independent of the specific strategies used in individual solvers, yet it gives key insights into the "progress" and "convergence behavior" of the solver in terms of a specific implementation at hand. An analysis tool is written to interpret the profile data and extract values of the following metrics such as: average convergence rate, efficiency of iteration and variable stabilization. Finally, using this system we produce a study of 4 well known SAT solvers to compare their iterative efficiency using random as well as industrial benchmarks. Using the framework, iterative inefficiencies that lead to slow convergence are identified. We also show how to fine-tune the solvers by adapting the key steps.

We also show that the similar profile data representation can be easily applied to loops, in general, to capture their program state. One of the key attributes of the program state inside loops is their branch behavior. We demonstrate the applicability of the framework by profiling completely parallelizable loops (no cross-iteration dependence) and by storing the branching behavior of each iteration. The branch behavior across a group of iterations is important in devising the thread warps from parallel loops for efficient execution on GPUs. We show how some loops can be effectively parallelized on GPUs using this information.

CHAPTER 1

INTRODUCTION

1.1 Background

Since it is not enough to have a computer program only being functionally correct but also have good performance characteristics, understanding and tuning the performance of the program becomes very important. The performance can be substandard because of (1) poor algorithm that is used to solve problem or (2) imperfect implementation and underlying architecture matching. In this work, we present a profile based framework to performance analyze both of the above problems and give suggestions for performance improvement.

Though it is true that most of the problems solved by computer have polynomial time solution, it is also true that a number of important problems do not have a polynomial time solution yet. The problems which do not have polynomial time solution are called NP (Non-deterministic polynomial time) and hardest set of problems in NP are called NP-Complete. Since, the complexity to solve most NP problems is exponential, as the problem size increases the time to solve the problem increases exponentially. Fortunately, for practically occurring instances of these problems, super-polynomial heuristics exist that allow solutions to be found in a reasonable amount of time. Since these algorithms are based on heuristics it becomes important to find the heuristic that work well for a given domain. The profile data to be analyzed to find a good heuristic can become very large, producing a need for a sophisticated analysis framework. In this case, since the data to be used to form heuristics is based on algorithm characteristics, it

is necessary to capture the state of algorithm and not the state of program. Our framework serves the purpose and captures the state of algorithm, which can be further used to find bottlenecks and tune the performance of NP problem solvers for particular application domain. We use SAT Solvers to demonstrate the applicability of our framework (more details on SAT Solvers are provided in chapter 2). We chose SAT Solvers to demonstrate applicability of our framework because all NP problems can be reduced to SAT Solving and other NP-Complete problems may not be optimally solved. In many cases SAT Solving can be extremely time consuming and many SAT Solvers time-out without producing an acceptable solution. As per [7], some SAT problems, which are on the SAT/UNSAT boundary, can be very time-consuming (one of the areas where these problems can be found is statistical physics analysis). This leads to the need of finding specialized heuristics for particular application domain, referred as tuning of SAT Solver. The other solution to improve performance of the solver is to, take advantage of multi-cores and parallelize the solver. But, the state-of-art of SAT algorithms are inherently sequential. There are some algorithms similar to [7], which can be parallelized but there are still many issues like handling of shared data etc.

The second case where the performance analysis becomes important is due to the advent of heterogeneous multi-cores. In recent years it is been shown that, accelerators like Graphics Processing Units (GPUs) can be used to improve the performance of applications significantly. GPUs have peculiar architecture characteristics and non-conventional programming model. These characteristics of GPUs can become big hurdles while porting existing code to GPUs [22]. Though it is now well understood that it is necessary to revise the algorithm to get maximum performance while porting programs to GPUs, there are many instances where the program can be ported to GPU without any algorithmic change. These program segments can be easily found as completely parallelizable loops annotated by OpenMP directives (*parallel for)*. Each iteration can be given to one GPU thread. In [22], authors demonstrate a compiler framework to

automatically translate OpenMP program segments into CUDA [23] (A GPU Programming Model by NVIDIA). There are two main problems in directly (without any algorithmic change) porting the OpenMP loops to GPU and getting performance improvement, (1) Branch Divergence (2) Uncoalesced Global Memory Accesses. In this work we develop a profile-based framework to extract information about branch divergence in parallel loops without actually porting to GPUs and provide hints to avoid those branch divergences if possible. (More details on GPU architecture are provided in chapter 6)

1.2 Thesis Contributions

Though numerous algorithms are developed for solving satisfiability problem, these different SAT solvers are compared and analyzed by using the execution time taken to solve a particular problem. There is a need for machine independent technique to provide an insight into the "progress" and "convergence" behavior of the solver. By knowing the iterative efficiency one can gain an understanding of the sources of inefficiency of the solver. The designer of the solver can then focus his attention on the specific phases and optimize performance of those phases. The ability to answer detailed questions about a given solver without knowing its internals also allows one to quantitatively compare different solvers; often time new solvers are concocted by mixing the best features of existing solvers. Our approach is to design a generic representation using performance data along with a set of metrics that helps to answer some specific questions by combining the metrics. Some representative questions are: to find out where a solver spent the maximum time? Which was the variable that was incorrectly assigned the value and that caused backtracking? This work presents a framework that supports comprehensive and generic set of metrics that are useful in performance analysis of various existing as well as future solvers. Our framework is useful in following aspects: 1) Performance debugging SAT solvers in an easiest way and 2) Building a performance comparison model by combining results of two or more metrics. This performance comparison model is targeted towards two distinct SAT communities: 1) Users of the SAT solvers, for choosing the most appropriate solver which suits their domain, and 2) Inventors of SAT solvers for performance debugging and for overcoming performance bottlenecks. We achieve this by first profiling the SAT solver through simple APIs that monitor key data-structures and provide the information about the variables and iterations. We then propose a set of metrics that can be efficiently composed to solicit

answers to the efficiency of a solver. Thus, an important contribution of this work is that it devises a framework which is completely independent of the specifics of the underlying solver and which can be generically applied to any solver. Moreover, the user is able to construct different quantifiable measures by composing a set of metrics we provide. Using these measures, a user can choose amongst the solvers and can then fine-tune a chosen solver.

In addition, we successfully apply the profiling techniques used for SAT Solvers to other general-purpose applications. We demonstrate this on applications that contain completely parallelizable loops without any loop carried dependencies. We profile the loops using Pin [27] dynamic instrumentation tool to gather information about the way in which branches behaved during each iteration. We use this profile information to find the loops suitability for Graphics processors (GPUs). More details about the technique and its usefulness for performance improvement of loops are given in chapter 7.

1.3 Thesis Organization

The rest of the thesis is organized as follows: chapter 2 gives an in-depth idea the boolean satisfiability problem and SAT Solvers, chapter 3 explains the framework and approach for SAT Solver debugging in detail, chapter 4 describes the case studies of several SAT Solvers using our framework, chapter 5 discusses the applicability and usefulness of our framework for SAT Solvers, chapter 6 provides background about GPU architecture, chapter 7 demonstrates the applicability of profiling framework for general purpose applications containing fully parallelizable loops, chapter 8 mentions related work, chapter 9 concludes the thesis and chapter 10 briefly discusses future work.

CHAPTER 2

BOOLEAN SATISFIABILITY PROBLEM AND SAT SOLVERS

This chapter gives a comprehensive background about Boolean satisfiability problem and SAT Solvers. The most prevalent SAT Solvers and their techniques are also discussed in detail.

2.1 Boolean Satisfiability Problem (SAT)

Computer Scientist has identified several important problems in computer science and Boolean Satisfiability problem (SAT) is one of them. It is important and gained lot of attention from researchers because many problems from real world can be easily reduced to SAT and till date, no polynomial time solution has been found for this problem. SAT Solver is being studied for more than 40 years. Moreover, the Boolean Satisfiability problem (SAT) is NP-Complete.

Given a conjunctive normal form (CNF) F specified on a set of variables {x1... xn}, the satisfiability problem is to satisfy (set to 1) all the disjunctions of F by some assignment of values to variables from {x1... xn}. A disjunction of F is also called a clause of F. Many problems such as microprocessor verification [1], logic synthesis [2], equivalence checking [3, 4], model checking [5] and FPGA routing can be easily reduced to the satisfiability problem (SAT). Moreover, SAT solving is very useful in various areas such as artificial intelligence, hardware design, electronic design automation and verification. SAT solvers also typically form the key element of combinatorial optimization packages such as CPLEX [14]. In the next section we show how a

combinatorial equivalence-checking problem can be represented as SAT problem in terms of CNF.

2.2 Combinatorial Equivalence-checking problem as SAT

Verification of implementation is an important issue in hardware design. Equivalence checking technique is used to verify the implementation. The Combinatorial Equivalence Checking (CEC) [21] is a fundamental technique for the equivalence checking for digital devices. The aim of CEC is to verify the equivalence of two combinatorial circuit designs; this is an important problem in hardware design and optimization.

2.2.1.1 Representing Combinatorial Circuit as SAT Problem

According to [21], the gate level language can be translated to CNF formula as following: Gates AND, OR, NOT can be translated to CNF formula as following.

For AND gate with *n* inputs, such as $out = AND (x_1, x_2, x_3, \ldots\ldots x_n)$ will generate n+1 clauses.

$\neg x_1 \cup \neg x_2 \ldots\ldots \cup \neg x_n \cup out$

$\neg out \cup x_1$

$\neg out \cup x_2$

.

.

$\neg out \cup x_n$

For OR gate with *n* inputs, such as $out = OR (x_1, x_2, x_3, \ldots\ldots x_n)$ will generate n + 1 clauses

$\neg x_1 \cup out$

$\neg x_2 \cup out$

$\neg x_3 \cup out$

.

$.\neg out \cup x_1 \cup x_2 \ldots\ldots \cup x_n$

For NOT gate, such as *out = NOT (x)*, will generate 2 clauses,

$out \cup x$

$\neg out \cup \neg x$

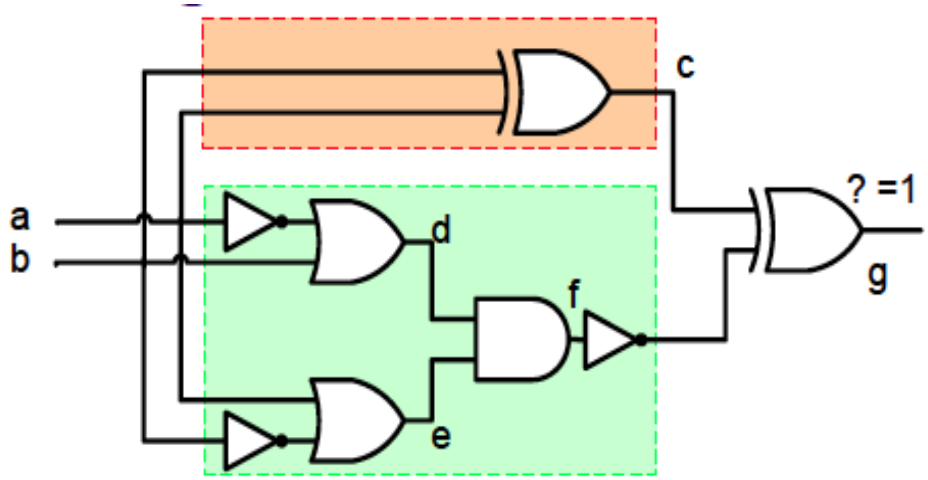

Figure 1: Combinatorial Circuit

The CNF formula for above circuit is as follows:

$(\neg a \cup \neg b \cup \neg c)(a \cup b \cup \neg c)(a \cup \neg b \cup c)(\neg a \cup b \cup c)$

$(a \cup d)(\neg b \cup d)(\neg a \cup b \cup \neg d)$

$(\neg a \cup e)(b \cup e)(a \cup \neg b \cup \neg e)$

$(d \cup \neg f)(e \cup \neg f)(\neg d \cup \neg e \cup f)$

$(\neg c \cup f \cup \neg g)(c \cup \neg f \cup \neg g)(c \cup f \cup g)(\neg c \cup \neg f \cup g)$

(g)

Using the above translation method, combinatorial logic can be translated into CNF formula. To check whether particular property P is always true for the gate circuitry, negation of that property is added to clauses. If there still exists a solution α then it is

confirmed that P is not always true and the solution α serves as counter example. For checking equivalence between two combinatorial circuits all the properties are checked one by one using SAT solvers and then they are proved as equivalent.

2.3 SAT Solvers

Due to its NP-complete nature and popularity, numerous algorithms are developed to solve the satisfiability problem in a fast and tractable way. The solver which solves this Boolean Satisfiability problem (SAT) is often referred as SAT Solver. The most prevalent classes of algorithms are modern variants of the DPLL (Davis-Putnam-Logemann-Loveland) algorithm such as zChaff [17], MiniSAT [16], HaifaSAT [13] and stochastic local search algorithms such as WalkSAT [12] or GSAT. Recently, as shown in [7], the techniques such as belief propagation and survey propagation are also used in finding solution to SAT problem especially for the set of problems on the boundary of SAT and UNSAT, so called hard SAT problems. Following is the brief description of SAT Solvers which are used in this thesis.

2.3.1 DPLL based Solvers

The basis of the DPLL algorithm is backtracking. It chooses a variable, assigns a truth-value to it and then keeps on recursively checking if the simplified formula is satisfied. If the check is successful then the original formula is satisfied. Otherwise, the same recursive check is done by assigning the chosen variable an opposite truth-value. The simplification step removes all clauses, which become true after the assignment of variable and all the variables that become false from the remaining formulas. DPLL enhances over the backtracking by unit propagation and pure literal elimination. Figure 2 shows the basic framework of DPLL algorithm.

```
while(1) {
  if (decide_next_branch()) { //Branching
        while(deduce()==conflict) { //Deducing
        blevel = analyze_conflicts(); //Learning
        if (blevel < 0)
            return UNSAT;
        else
            back_track(blevel); //Backtracking
        }
  }
  else //no branch means all variables got assigned.
        return SATISFIABLE;
}
```

Figure 2: DPLL Algorithm Framework

Most modern algorithms use the DPLL framework and differ in the implementation of branching heuristics (`decide_next_branch()`), deduction algorithm (`deduce()`), conflict analysis (`analyze_conflict()`) and backtracking (`back_track()`) functionalities. Following is the description of DPLL based solvers based on above-mentioned four functionalities.

2.3.1.1 zChaff

zChaff uses Variable State Independent Decaying Sum (VSIDS) branching heuristic. In this strategy variables are ranked by number of occurrences of the

literal in the initial clause database for a problem. These occurrence counts are incremented when new clauses are added and periodically divided by a constant. VSIDS score is a literal occurrence count with more weight on most recently added clauses. For branching, a free variable is chosen with highest combined score. This decision strategy gives more importance to variables that were involved in recent conflicts. zChaff uses 2-literal watching algorithm for deduction. The algorithm works in following way:

1. Each clause has two watched literals that initially free can move in any direction.
2. Each variable keeps two linked lists that contain pointers to all the watched literals: one for positive watched literals and second for negative watched literals of that variable.
3. When variable *v* is assigned 1, for each literal *p* pointed by a pointer in the list of neg_watched (v) the solver searches for a literal *l* in the clause (containing *p*) that is not set to 0. Similarly, if variable *v* is assigned 0 it searches in pos_watched (v) list.

   Following four cases exists while assigning a value to variable:

   a. If the only such literal *l* is other watched literal and it evaluates to 1, then clause is satisfied, hence do nothing.
   b. If there exists such literal *l* and it is not the other watched literal then remove p from neg_watched (v) and add pointer to *l* to the watched list of the variable corresponding to *l*. Move the watched literal for that clause to *l*.
   c. If only such *l* is the other watched literal and it is free then the corresponding clause is unit clause and other watched literal is unit literal.
   d. If all literals in clause become 0 then it is a conflicting clause.

zChaff uses all/first unique implication point conflict analysis for learning and backtracking. For more details on unique implication point conflict analysis refer to [19].

2.3.1.2 Minisat

The branching heuristic of Minisat is an improved VSIDS order, where variable activities are decayed 5% after each conflict. The original VSIDS decays variables 50% after each 1000 conflicts or similar. It uses similar algorithm of that is used in zChaff but it processes binary clauses differently. Binary clauses are implemented by storing the literal to be propagated directly in the watcher list. Minisat also uses Conflict driven learning and backtracking scheme.

2.3.1.3 HaifaSat

HaifaSAT follows the abstraction/refinement model and developers of HaifaSAT view SAT Solver as proof engine than search engine. The algorithm used for conflict analysis and back tracking is mainly based on First Unique Implication Point algorithm and it is called as Variable Move To Front (VMTF).

2.3.2 Stochastic Local Search Algorithm based Solvers

In local search algorithms solution space is viewed as a set of points connected to each other. For selection of variable and its assignment a cost function is defined and it has to be computed at the end of every iteration. The variable should be selected such that cost is minimum. Local search algorithm starts at some point in solution space and moves to adjacent spaces in an attempt to lower cost function. The search is said to be greedy if it does not ever increase the cost function (it may lead to local minima).

2.3.2.1 WalkSAT:

The algorithm specified in [12] starts by randomly assigning a value to each variable, if the assignment satisfies all the clauses then algorithm terminates by returning the assignment. Otherwise it picks a clause randomly and flips one of its variables; the

variable is selected such that it will result in fewest of previously satisfied clauses becoming unsatisfied.

CHAPTER 3

FRAMEWORK FOR CHARACTERIZING SAT SOLVERS

Our approach is divided into various phases: instrumentation phase, profiling phase, analysis phase, each of which forms a component in our framework. The rest of this section explains each phase in detail.

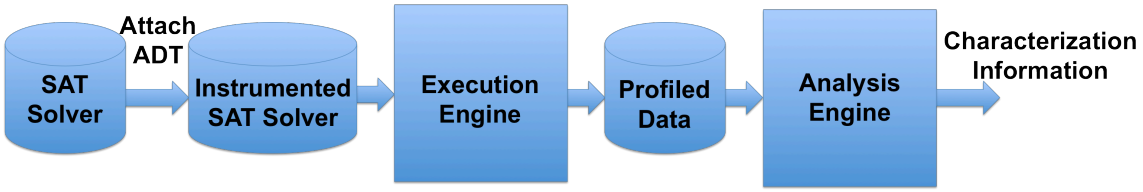

Figure 3: Framework

3.1 Instrumentation

This phase is responsible for modifying the solver and attaching the instrumentation code to it. When this instrumented solver is executed it will generate the profile data containing value assigned [True, False, Undef] to every variable for all iterations executed. The Instrumentation is done at source code level and it is expected to be done by the implementer of the algorithm. Before finalizing manual source level instrumentation strategy, we tried various automated and binary level instrumentation strategies but then we found that these strategies were not at all feasible, as we are trying to gather the algorithm characteristics and not the program characteristics. Manual source level instrumentation is difficult and could generate non-uniform profile data. To solve this problem we have designed and implemented an abstract data type, which makes this job fairly easy for the implementer.

```
void StoreProfileData(int NumberOfVariables)
void StartTimer();
void SendRestartSignalToProfiler();
void SetDecisionVariable(int Variable);
```

Figure 4: Prototype of Instrumentation APIs

Implementer has to provide the location of all the variables either in terms of array, list, vector, etc. and call our `StoreProfileData` function at the end of each iteration. This function takes the snapshot of the values assigned to all the variables at that instance and stores it in our internal data structure (This data structure is explained later in this chapter). To facilitate the calculation of time taken by each iteration, the `StartTimer` function needs to be called at the start of every iteration. This function in combination with `StoreProfileData` is responsible for approximating the time required for each iteration to complete.

There are other supporting functions like `SendRestartSignalToProfiler` and `SetDecisionVariable` that are used to collect the information about internal state of the solver. The `SendRestartSignalToProfiler` function is called whenever the solver performs restart operation. The `SetDecisionVariable` is used to know the decision variable for that iteration.

The `PostProcessData` function must be called before the SAT solver finishes, which is responsible for dumping the summery profiling data into the file, which will be used by Analysis phase, described in section 3.3.

3.2 Profiling SAT Solver

In this phase, the instrumented SAT solver is executed by specifying the CNF formula as input. This CNF formula should, in general, represent the application domain for which the SAT solver is going to be used. Every iteration is profiled, which stores value of each variable and additional information such as execution time spent, decision variable and difference vector. These values are then dumped into the file at the end of the execution.

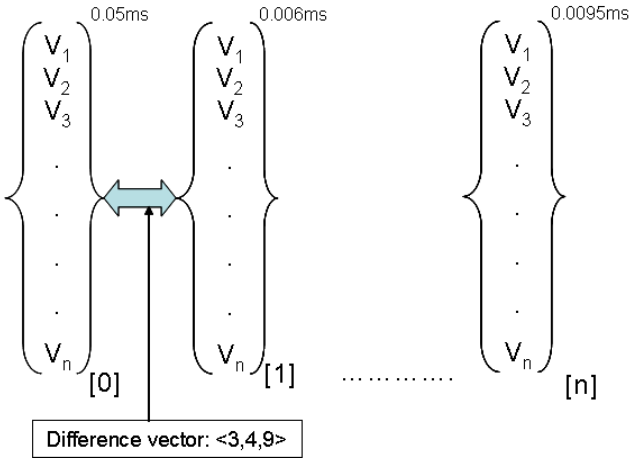

Figure 5: Internal Representation of Profiled Data

As explained later in section 3.3, this extra information is useful in calculating the convergence rate, stabilization iteration etc. Figure 5 shows the internal representation of profile data. Each column represents data for iteration. For each iteration, a difference vector is added, which contains the variables whose values are changed from previous iteration to current iteration. Also with respect to each iteration, information about time taken to execute that iteration is added. As an example, as per figure 5, the iteration number are 0, 1 till n. The difference vector <3, 4, 9> tells that variable number 3, 4 and 9 have been modified from iteration 0 to 1. The floating-point numbers at the top

represent the time spent in that iteration. As the number of iterations required by SAT Solver to obtain the solution can be very large, the above profiling scheme can generate data upto 5-10 GBs (solver executing for approximately 3 to 4 hours). So, to reduce the profiling data size we propose two other schemes 1) Selective Profiling 2) Compact Profiling.

3.2.1 Selective Profiling

In this scheme of profiling, the user can select the kind of profile data to be generated by our tool. The selective profiling scheme is dependent on the number of propagations taking place in that iteration. The user has to provide a condition on number of propagations allowed per iteration to be included in profile data. The iterations will be selected depending on the number of propagations specified and the condition on it. The condition could be of two types 1) greater than the number of propagations specified by user 2) less than the number of propagations specified by user. This scheme helps user to generate selective profile data, which can be easier to comprehend and uses lesser disk space.

3.2.2 Compact Profiling

This profiling scheme is based on storing the data in a form that is compact, but requires a much more sophisticated analysis tool than required by previously mentioned techniques. In this scheme only the difference vector and decision variable for each iteration is stored. Storing only the difference vector makes this scheme candidate for requiring a sophisticated analysis tool. For analyzing state of all the variables at particular

iteration, it needs to scan all previous iteration. This representation helps to reduce the profile data size significantly.

3.3 Analysis Phase

The operation of this phase is totally in isolation from previous two phases. This phase facilitates analysis of solver behavior offline. In this chapter, first we describe metrics developed by us to characterize SAT Solvers and then explain functionality of Graphical User Interface based analysis tool.

3.3.1 Metrics

The metrics defined in this section are generic enough to be applied to any type of SAT Solvers mentioned in section 2.3.

1) **Stabilization iteration of given variable:** This query can be useful if user wants to compare the stabilization iteration variable V_i with stabilization iteration of variable V_j. The stabilization iteration of a variable is the iteration where its value was last assigned during the execution.

2) **Average time spent per iteration:** This query finds average time spent per iteration.

3) **Extract Decision Variables:** Decision variables are the variables whose assignment was not forced by values of other variables and this initial assignment did not cause any backtracking and included as it is in the solution of the formula. This query extracts variables with unforced stable assignment and returns a set.

4) **Correlation between variables:** This query specifies correlation in terms of decision variables and implied variables. Implied variables are the variables whose final assignment was forced by values other variables. This query finds the

relation between decision variables to implied variables. For a given decision variable v_1 (which can be found out by query (3)), it gives the set of variables denoted by *Implied* (v_1) where final assignment of each variable in *Implied* (v_1) is forced by v_1.

5) **Variables stabilized by iteration *k*:** This query gives the number of variables stabilized in or before iteration *k*. The stabilization iteration of the variable is the iteration in which final value is assigned to that variable which is included in the solution.

6) **Variables stabilized in given iteration interval:** This query gives the variables that were stabilized between two given iterations. Consider, if difference is need to be found in i^{th} and j^{th} iteration (j-i) then the resulting set will include the all variables stabilized up till j^{th} iteration but not stabilized up till i^{th} iteration

7) **Iterations spent in stabilizing given variable:** This query gives the iterations spent in stabilizing given variable. This is calculated by taking the difference of iteration number where the given variable v_1 last changed and the iteration number where the given variable v_1 was first changed.

8) **Average rate of convergence:** The average rate of convergence can be defined in three different ways. 1) Number of variables stabilized per iteration, 2) Average number of iterations required to stabilize a variable and 3) Average time required to stabilize a variable. This query is divided in to three sub queries according to 1, 2 and 3 to find the average rate of convergence as variables/iteration, iteration/variable and time/variable respectively. It internally uses query number (6) and (7).

9) **Restarts with condition on implications:** This query gives the restart numbers, which satisfy condition on implications. The condition can be *implications* > *x* or *implications* < *x*, where *x* is provided by analyzer.

10) **Iterations with condition on implications:** This query gives the iterations, which satisfy condition on implications. The condition can be *implications* > *x* or *implications* < *x*, where *x* is provided by analyzer.

3.3.2 Analysis Tool

This section describes the analysis tool and its features. The analysis tool uses compact profiling data format as its input. It shows the information about each iteration and the state of the solver at that iteration in terms of aggregated information. Following section describes features of the analysis tool in detail.

3.3.2.1 Features

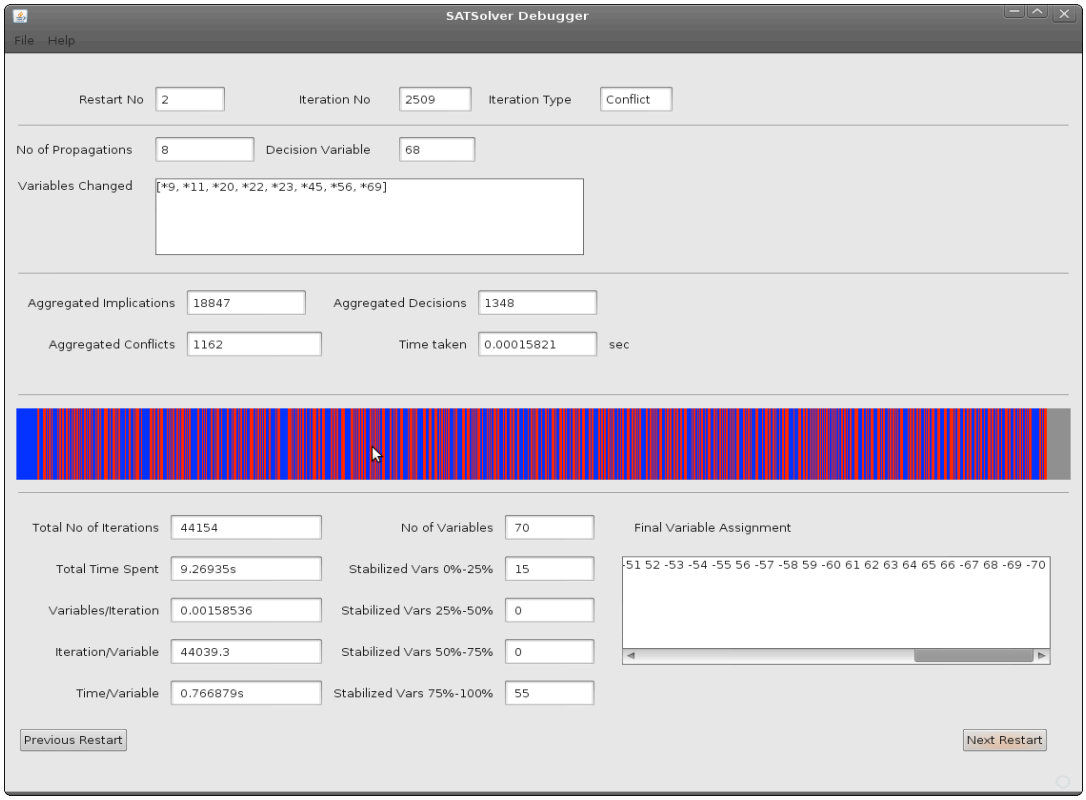

Figure 6: Snapshot of Analysis Tool

Figure 6 shows the snapshot of analysis tool. The fields are divided into Iteration Specific Fields and Execution Summery fields. The information about each field is as follows:

Iteration Specific Fields:

These fields are specific to each iteration and are changed depending on the iteration selected using progress indicator grid.

1) **Restart No:** This field displays restart number for iteration information currently displayed. This field is incremented and decremented as user presses button "Next Restart" and "Previous Restart" respectively. All other fields are populated with respect to Restart No field.
2) **Iteration No:** This field displays the iteration number currently active. All the other fields are populated with respect to this number in current restart. Clicking on progress indicator grid can change the iteration number.
3) **Iteration Type:** There are two types of iterations 1) Normal iteration 2) Conflict iteration. Conflict iterations are the iterations in which conflict was occurred. All other iterations are Normal iterations.
4) **No of Propagations:** This field displays the number of propagations happened during that iteration. This can be used to judge progress of solver during that iteration. It shows the number of variables changed from previous iteration.
5) **Decision Variable:** This field displays the decision variable for that iteration. For conflict iteration this will show the last variable assigned which is responsible for the conflict.
6) **Variables Changed:** This field displays the variables that are changed during that iteration and the values assigned to them for that iteration. For example, [49, -282] represents that Variable 49 and Variable 282 were changed. The variable 49 was assigned value True and variable 282 was assigned value False. A '*' before

the value means that variable was reset during backtracking and it is undefined. The '*' can be seen before the variable number in conflict iteration.

7) **Aggregated Implications:** This field shows the sum of all implications that happened till that iteration.
8) **Aggregated Decisions:** This field shows the sum of all decision happened till that iteration.
9) **Aggregated Conflicts:** This field shows the sum of all conflicts happened till that iteration.
10) **Time taken:** This field shows the time taken by that iteration. This field will be helpful to know the time spent in each iteration. This will be helpful to analyze variation in time because of variety of conflict and deduce strategies.
11) **Progress Indicator Grid:** This is a grid, which shows the iteration space with color layout. The "Blue" color represents normal iteration and "Red" color represents conflict iteration. User can click on the grid to select the iterations and all the fields will be repopulated to show the information of respective iteration.

Execution Summery Fields:

These show summery of solver execution. These fields are actual metrics described in section 3.3.1 or derived from actual metrics.

12) **Number of Variables:** This gives the total number of variables in the problem.
13) **Total No of Iterations:** This field displays the total number of iterations required for solver to obtain the solution to given problem. These are the aggregation of iterations performed during each restart.

14) **Total Time Spent:** This field displays the total time spent for solver to obtain the solution to given problem.

15) **Variables/Iteration:** This field displays rate of convergence in terms of average variables stabilized per iteration.

16) **Iterations/Variable:** This field displays rate of convergence in terms of average number of iterations required to stabilize a variable.

17) **Time/Variable:** This field displays rate of convergence in terms of average time spent to stabilize a variable.

18) **Stabilized Vars 0%-25%:** This field displays the variables stabilized in 0% to 25% iterations of last restart. These values show the progress behavior of solver during those iterations.

19) **Stabilized Vars 25% to 50%:** This field displays the variables stabilized in 25% to 50% iterations of last restart.

20) **Stabilized Vars 50% to 75%:** This field displays the variables stabilized in 50% to 75% iterations of last restart.

21) **Stabilized Vars 75% to 100%:** This field displays the variables stabilized in 75% to 100% iterations of last restart.

22) **Final Variable Assignment:** This field displays the final Variable assignment for satisfiable problems.

23) **Next Restart:** This button if pressed displays the execution characteristics of next restart if available.

24) **Previous Restart:** This button if pressed displays the execution characteristics of previous restart if available.

CHAPTER 4

CASE STUDIES – SAT SOLVERS

4.1 Case Studies

We implemented our framework in C/C++ and used it to analyze the performance of 4 popular solvers; MiniSAT (v1.14), zChaff [10], WalkSAT (v46) and HaifaSAT (v1.0). Currently the only basis of comparison of these solvers is the total execution time they take on some standard inputs, as used in SAT competitions. Designers as well as users of these solvers are well served if comparisons on the key metrics mentioned earlier are provided. These can be very useful to choosing a solver as well as fine-tuning it. In this case study we first show a performance comparison of these solvers on the basis of the metrics mentioned in the previous section. Using these metrics, we expose some key performance bottlenecks in "underperforming" solvers and then propose new solutions to remedy them.

As an input we used one instance of 3SAT (containing 360 variables, 1533 clauses), one instance of 5SAT (containing 70 variables and 1491 clauses) and one instance of 7SAT (containing 45 variables and 4005 clauses) problem from "random" benchmark which was used in SAT 2007 competition, along with a simple 3SAT problem containing 100 variables, 400 clauses named f100 (included in WalkSAT source code package). We ran

our experiments on machine having Intel ® Xeon ® 2.67GHz and running Linux kernel version: 2.6.18.

We have analyzed these solvers using following measures, which use a combination of metrics described in Section 3.3.1.

1) **Total Number of iterations required:** This metric is used to find number of iterations required by the solver to find the solution. With reference to Table 1, we can see that zChaff requires least number of iterations for f100 and iteration count increases for 5SAT and 3SAT. It can be inferred from results that increase in number of variables affects zChaff's execution more than that of increase in variables per clause. Similar behavior can be observed for MiniSAT. In contrast with this, HaifaSAT's iteration count is affected significantly by increase in number of variables in clause than overall increase in variables. For WalkSAT, the iteration count for increase in number of variables or number of clauses does not differ by much.

Table 1: Total Number of Iterations required to obtain Solution

	F100	3SAT	5SAT	7SAT
zChaff	46	1669238	477298	407007
MiniSAT	216	1193923	408197	2819023
WalkSAT	8159	163312	86418	828310
HaifaSAT	92	815	44154	15155796

Table 2: Total time spent

	f100	3SAT	5SAT	7SAT
zChaff	0.000733589s	315.634s	92.1055s	177.034s
MiniSAT	0.00386333s	81.8592s	20.9032s	338.999s
WalkSAT	0.00726666s	0.160308s	0.215659s	5.67988s
HaifaSAT	0.00162715s	0.031855s	8.1771s	15636.6s

2) **Average time (seconds) spent per iteration:** This metric finds the average time spent per iteration using by using query number (2) described in section 3.3. The results are shown in Table 3. It can be observed HaifSAT and zChaff's time significantly increases as the number of variables per clause are increased. For HaifaSAT it gets worst for 7SAT problems. It can be also observed that, WalkSAT spends less time per iteration compared to others but as it can be seen from Table 1 it requires maximum iteration. Thus, it can be inferred that WalkSAT does the minimum calculation per iteration, which does not contribute significantly in solving the problem.

Table 3: Average Time Spent per Iteration

	F100	3SAT	5SAT	7SAT
zChaff	1.59476e-05s	0.000189089s	0.000192973s	0.000434965s
MiniSAT	1.78858e-05s	6.85632e-05s	5.12085e-05s	0.000120254s
WalkSAT	8.90631e-07s	9.81603e-07s	2.49554e-06s	6.85719e-06s
HaifaSAT	1.76864e-05s	3.90859e-05s	0.000185195s	0.00103173s

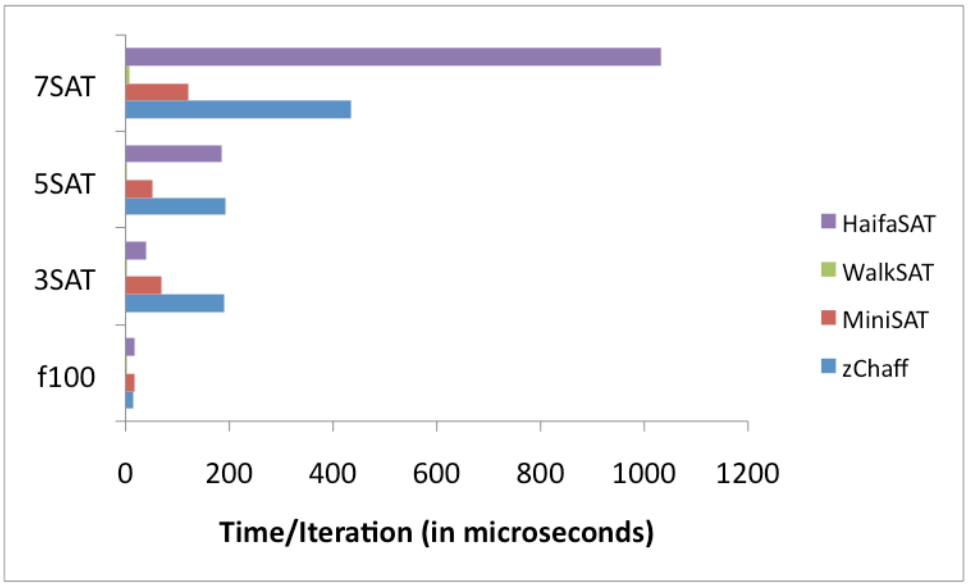

Figure 7: Average Time Spent per Iteration (in microseconds)

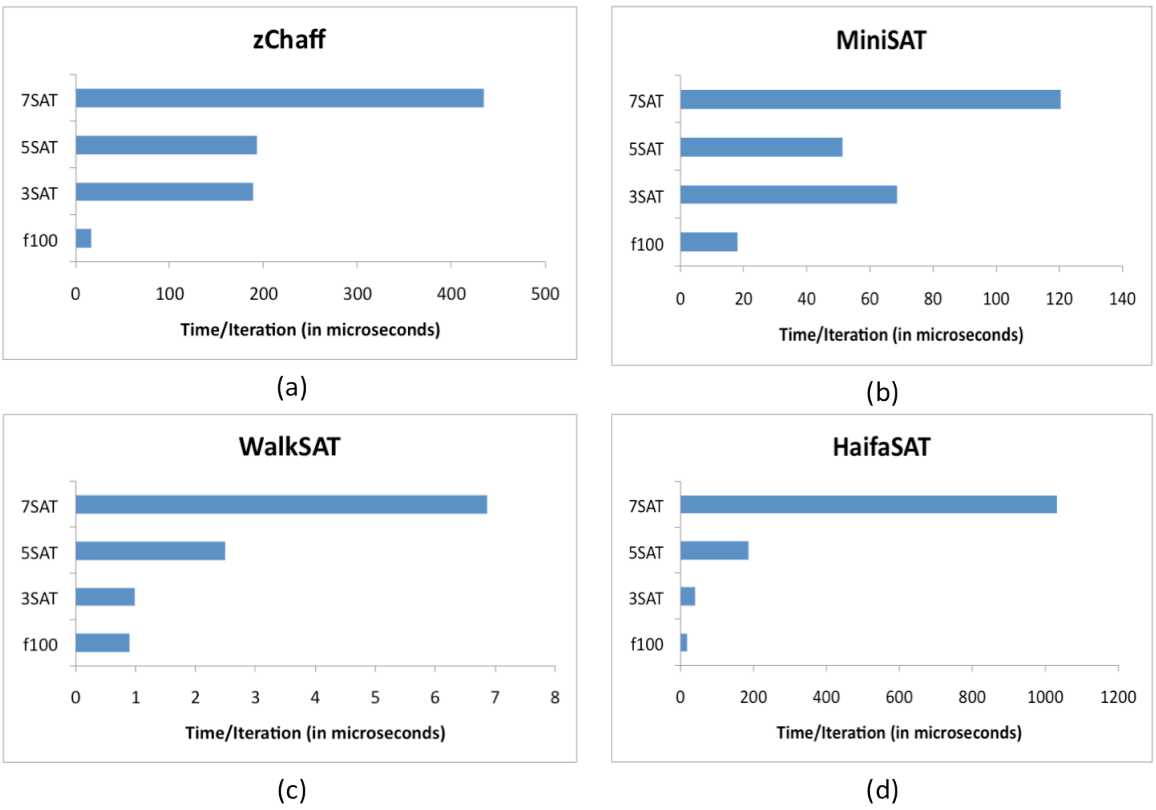

Figure 8: Average Time Spent per Iteration detailed (in microseconds) (a) zChaff (b) MiniSAT (c) WalkSAT (d) HaifaSAT

3) **Variables stabilized in first half of iterations:** This metric gives the number of variables stabilized in first half of iterations of last restart. Interestingly, only HaifaSAT is able to stabilize variables in first half of iterations. In case of 5SAT HaifaSAT is not able to stabilize any variables in first half of the iterations. So it can again be observed (as in Table 1 & 3) that HaifaSAT has problems in getting the solution quickly if the number of variables per clause increases.

Table 4: Number of Variables Stabilized in 0% to 50% of Iterations

	F100	3SAT	5SAT	7SAT
zChaff	25	70	10	13
MiniSAT	5	9	5	2
WalkSAT	17	32	0	0
HaifaSAT	17	60	15	11

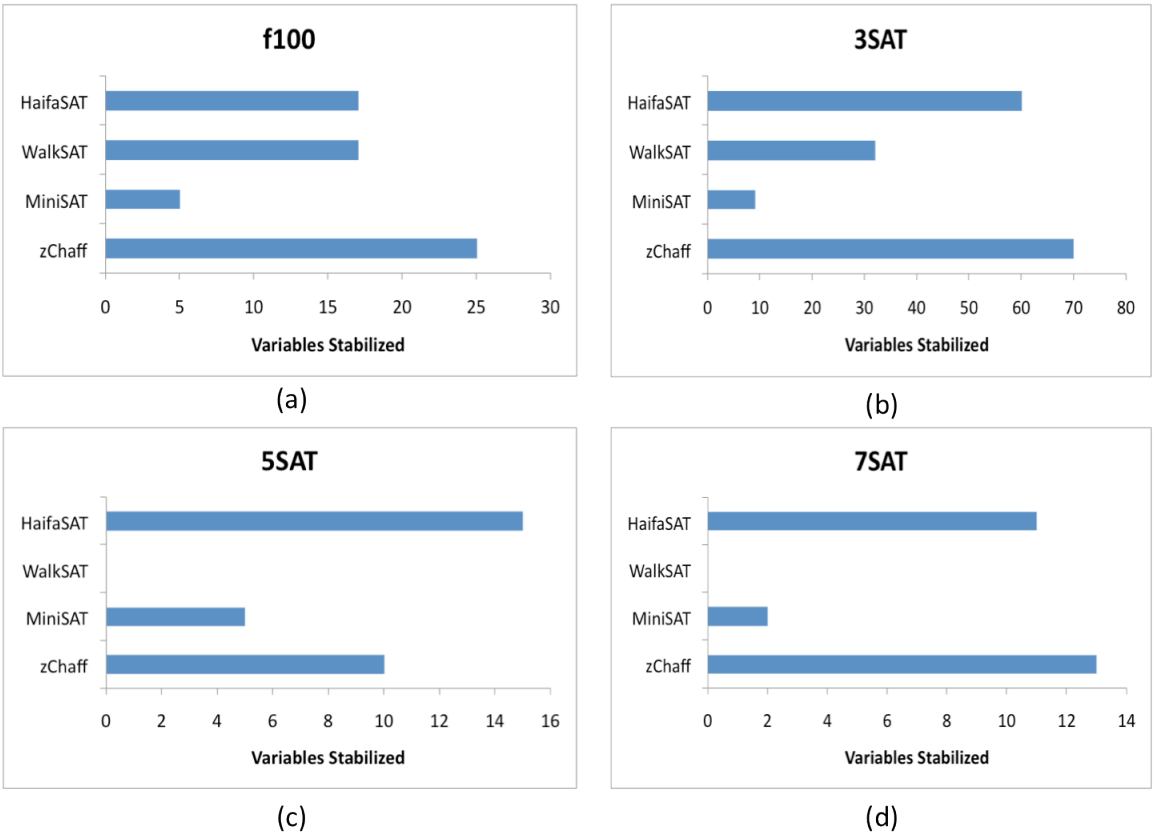

Figure 9: Variables Stabilized from 0% to 50% of Iterations (a) f100 (b) 3SAT (c) 5SAT (d) 7SAT

4) **Variables stabilized in 50% to 75% of iterations:** This metric gives the number of variables stabilized in 50% to 75%. As it can be seen from Table 4, there is no significant improvement in number of variables stabilized as compared to metric (3), though HaifaSAT manages to stabilize some more variables.

Table 5: Number of variables stabilized in 50% to 75% of the iteration

	F100	3SAT	5SAT	7SAT
zChaff	11	25	1	2
MiniSAT	1	1	1	1
WalkSAT	15	30	0	0
HaifaSAT	1	2	0	0

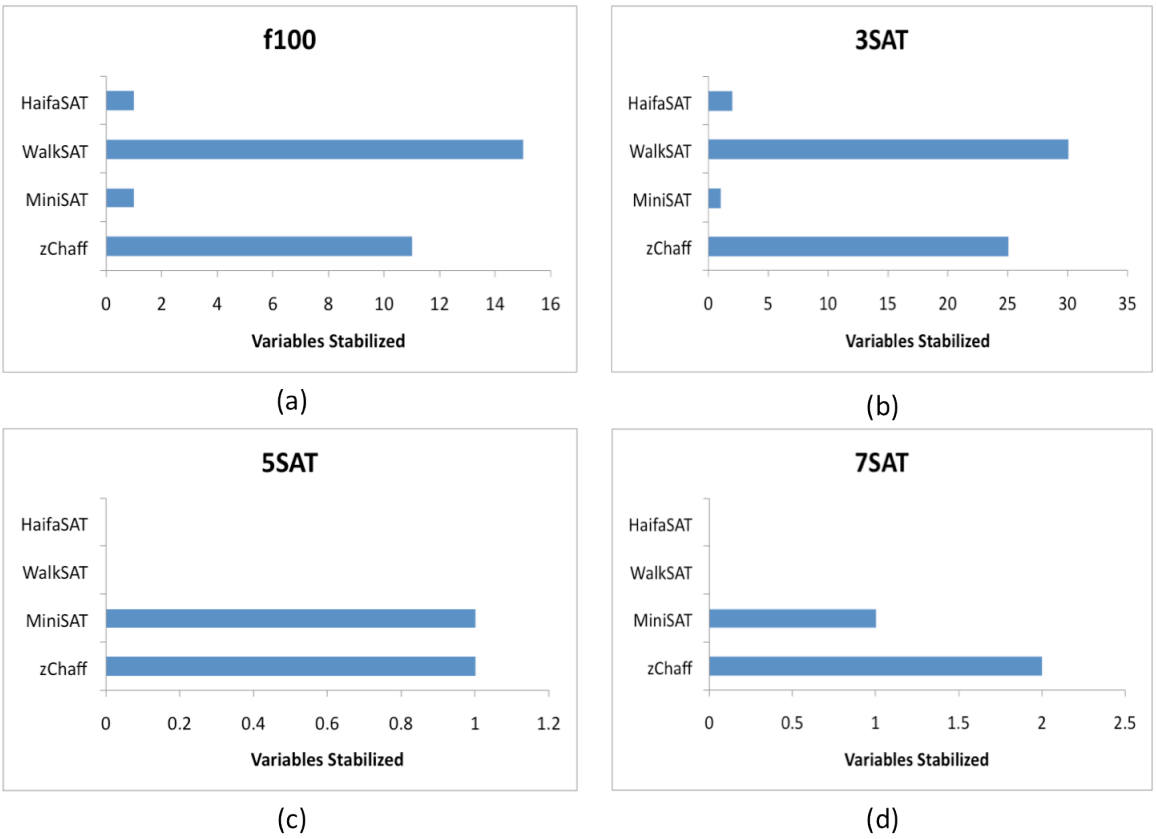

Figure 10: Variables Stabilized from 50% to 75% of Iterations (a) f100 (b) 3SAT (c) 5SAT (d) 7SAT

5) **Variables stabilized in 75% to 100% of iterations:** This metric gives the number of variables stabilized in the last quarter of the iterations. As it can be seen from Figure 4, for all solvers, most variables are stabilized in the last quarter of the iterations.

Table 6: Number of Variables Stabilized in 75% to 100% of the Iteration

	F100	3SAT	5SAT	7SAT
zChaff	64	263	59	30
MiniSAT	94	350	64	42
WalkSAT	64	296	70	45
HaifaSAT	82	298	55	34

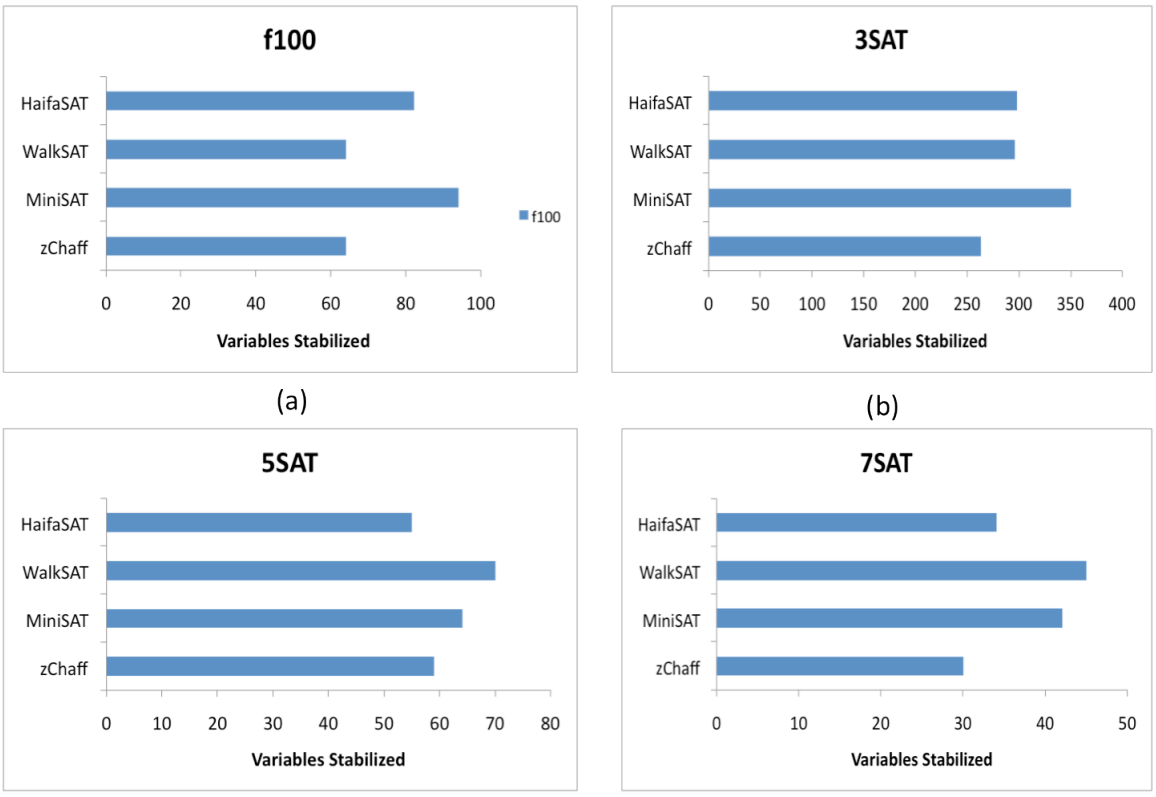

Figure 11: Variables Stabilized from 75% to 100% of Iterations (a) f100 (b) 3SAT (c) 5SAT (d) 7SAT

6) **Average rate of convergence (Variables/Iteration):** This metric specifies the fruitfulness of each iteration with respect to stabilizing the variable. It can be inferred from Table 7 that HaifaSAT's convergence efficiency per iteration is more than any other solver (except for f100 input), it means that HaifaSAT does more useful work every iteration and finds the solution in minimum iterations as well as time. Also, since results shown in Table 7 align with results in Table 1, we can say that, number of iterations required in finding the solution is an important parameter to be considered while calculating the efficiency of the SAT solver.

Table 7: Number of variables stabilized per Iteration

	F100	3SAT	5SAT	7SAT
zChaff	2.17391	0.000215667	0.000146659	0.000110563
MiniSAT	0.462963	0.000301527	0.000171486	1.5963e-05
WalkSAT	0.0122564	0.00220437	0.000810016	5.43275e-05
HaifaSAT	1.08696	0.441718	0.00158536	2.96916e-06

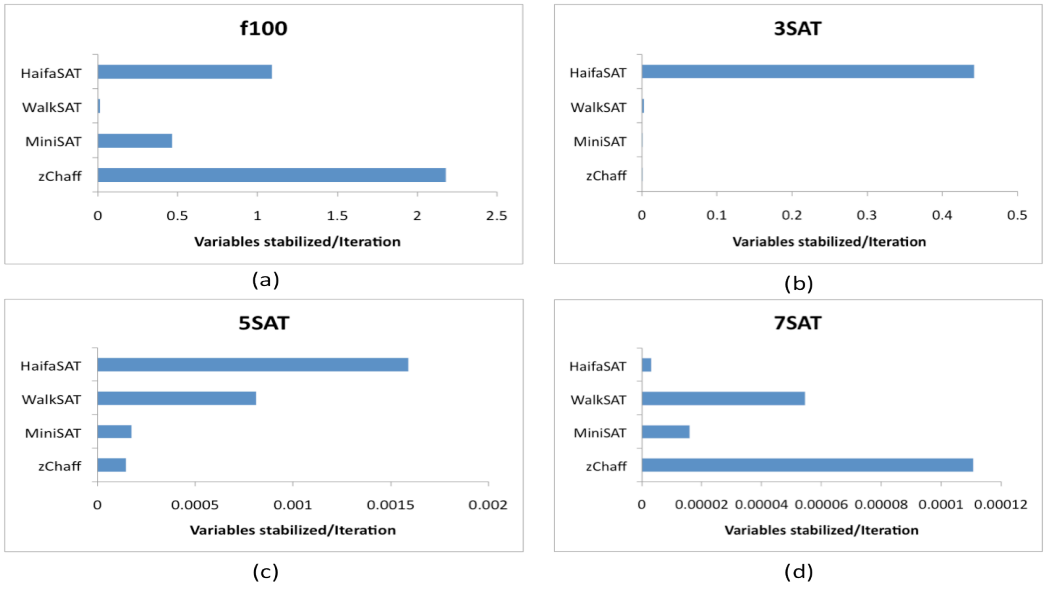

Figure 12: Number of Variables Stabilized per Iteration detailed (a) f100 (b) 3SAT (c) 5SAT (d) 7SAT

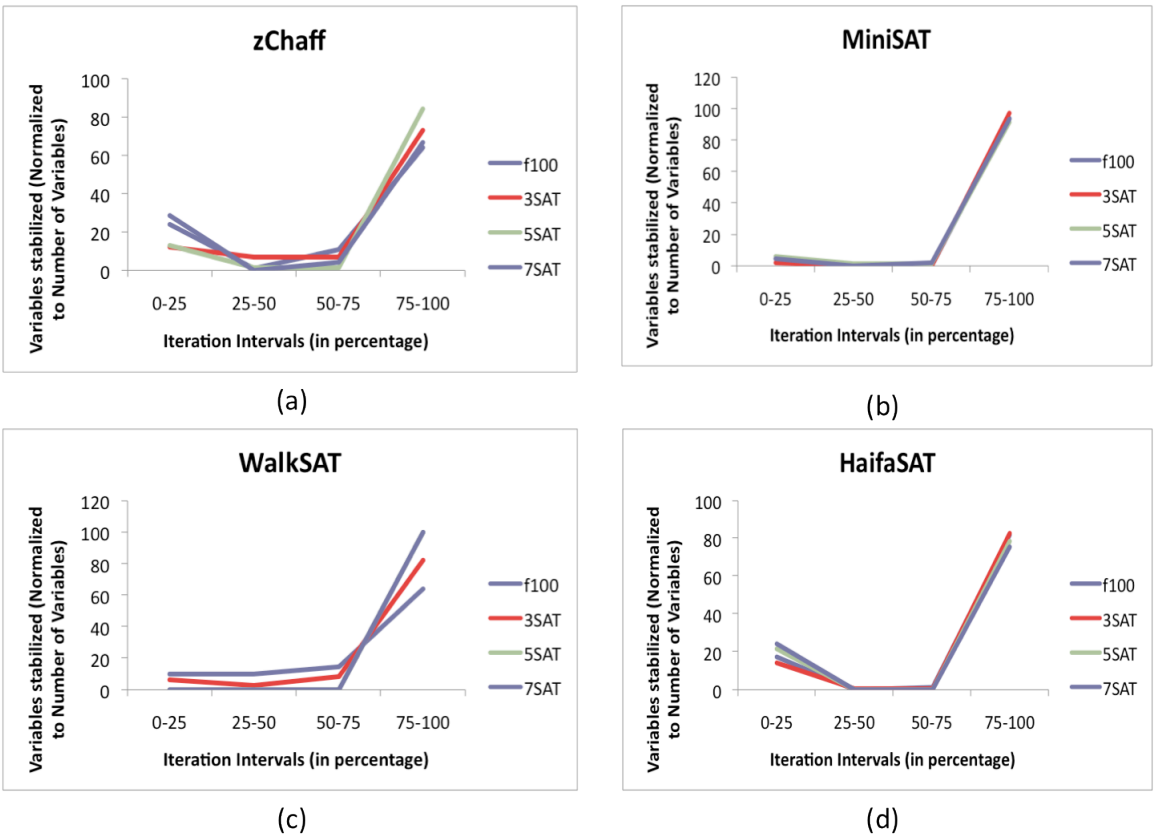

Figure 13: Stabilization of Variables - Comparison (a) zChaff (b) MiniSAT (c) WalkSAT (d) HaifaSAT

7) **Average rate of convergence (Iterations/Variable):** This metric gives the average number of iterations required to stabilize a variable. In this case also HaifaSAT outperforms other solvers in most cases.

Table 8: Average Iterations per Variable Stabilization

	F100	3SAT	5SAT	7SAT
zChaff	29.08	1.66915e+06	477177	406852
MiniSAT	119.79	1.18802e+06	398530	2.77132e+06
WalkSAT	7909.97	161070	86322.6	828266
HaifaSAT	27.96	492.936	44039.3	1.51555e+07

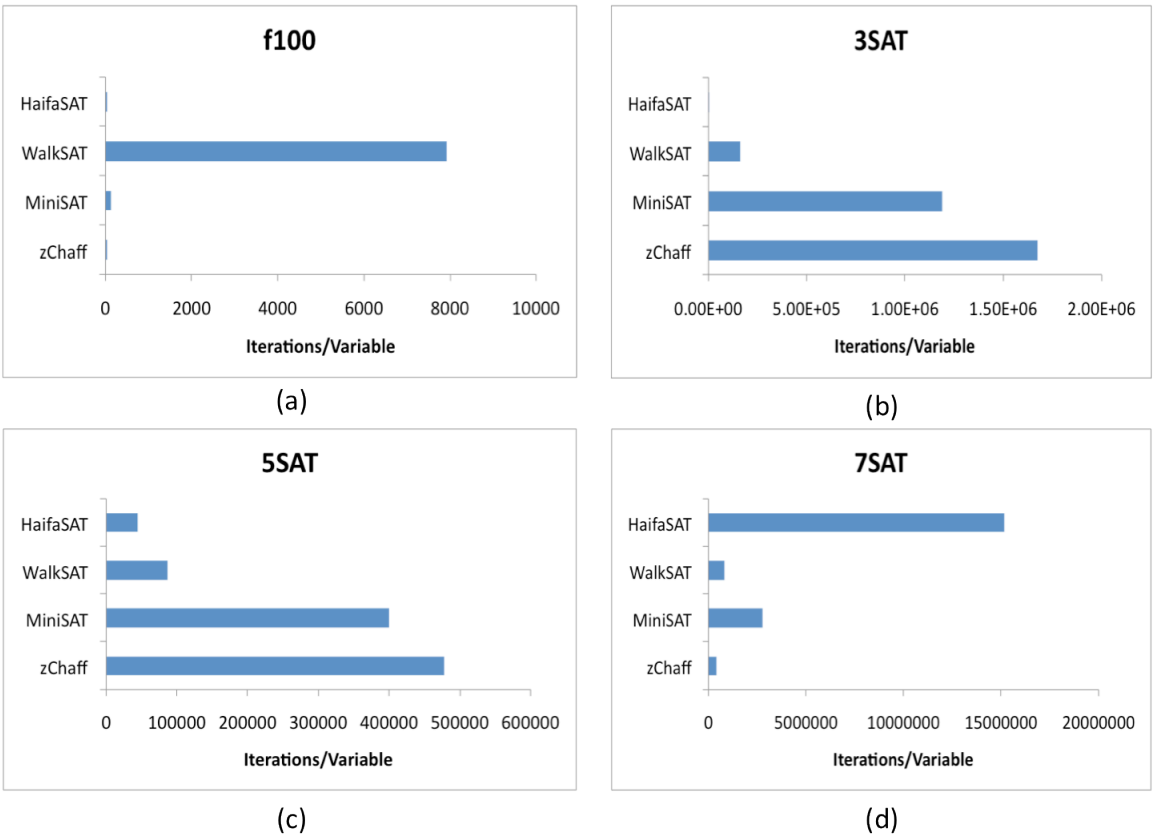

Figure 14: Average number of iterations taken for a variable to stabilize (a) f100 (b) 3SAT (c) 5SAT (d) 7SAT

8) **Average rate of convergence (Time (seconds) /Variable):** This metric gives the time required to stabilize a variable in seconds. It is similar to metric (6) but this metric is more useful to a user who is aiming to fine-tune an implementation on a given machine in addition to the algorithmic measure. It gives average time taken to stabilize a variable.

Table 9: Average Time taken per Variable Stabilization

	F100	3SAT	5SAT	7SAT
zChaff	0.000179538s	89.8549s	26.3595s	36.9431s
MiniSAT	0.000117259s	0.850955s	0.510283s	10.7787s
WalkSAT	7.91374e-05s	0.000452906s	0.00309631s	0.126258s
HaifaSAT	7.35771e-05s	0.000349316s	0.0820154s	163.894s

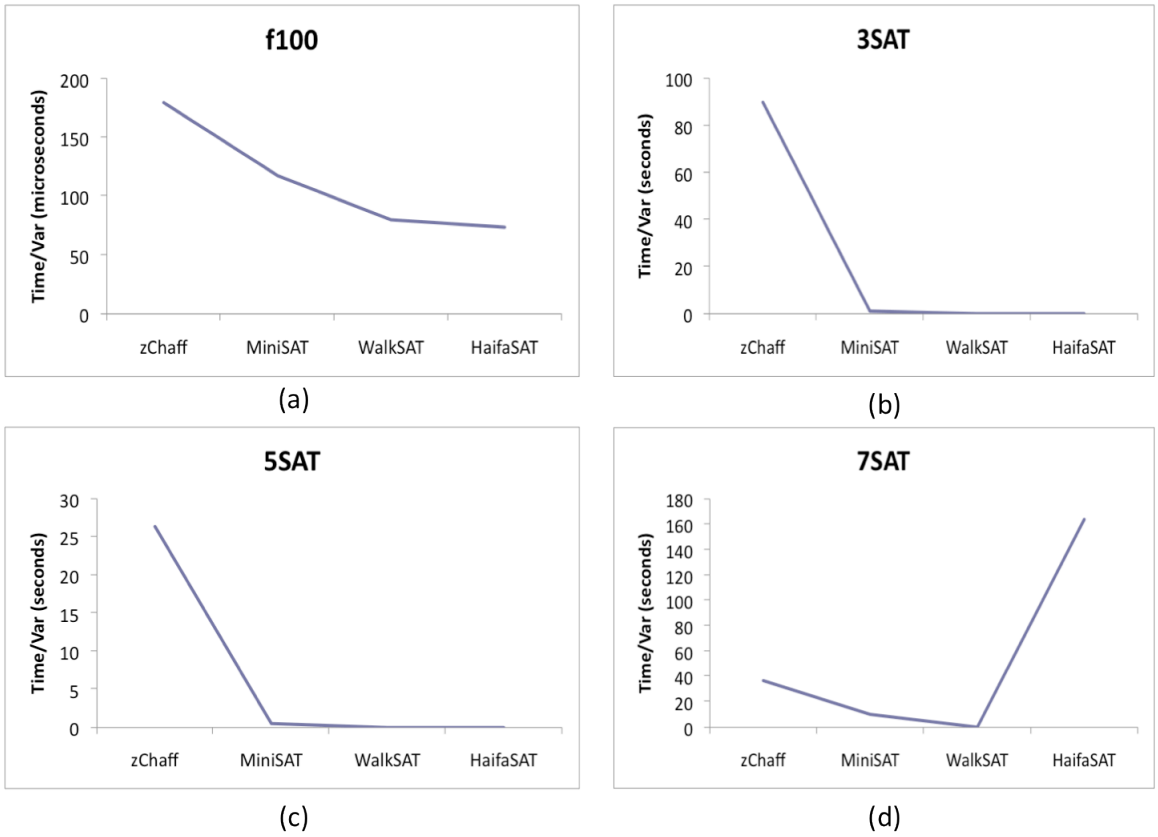

Figure 15: Average Time taken per Variable Stabilization (a) f100 (b) 3SAT (c) 5SAT (d) 7SAT

4.2 Profile Data Size Experiments

Here we present results of different profiling strategies we implemented in terms of size of profile data. As mentioned in section 3.2 there are three profiling strategies we implemented, following are the profile data sizes for HaifaSAT when run on 7SAT problems.

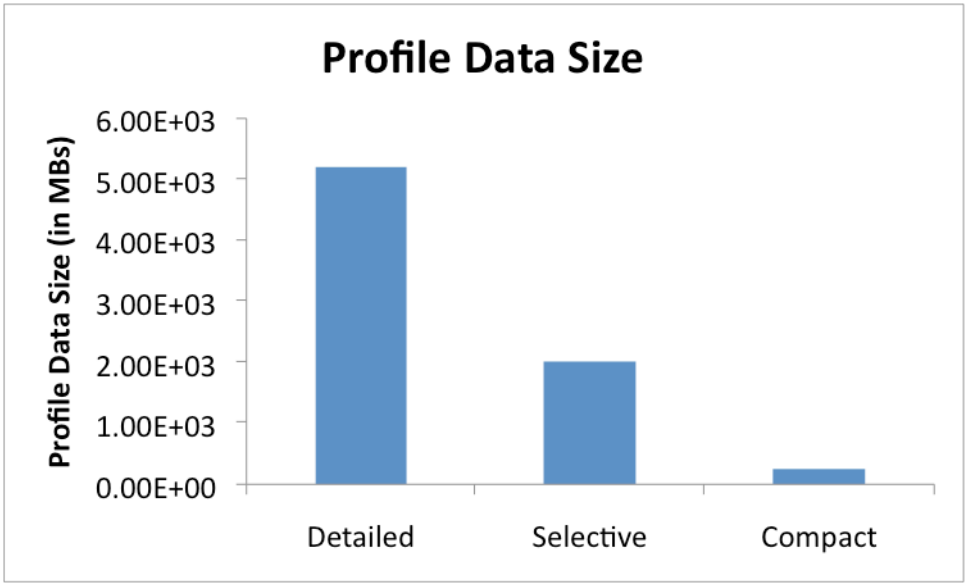

Figure 16: Profile Data Size with respect to various profiling techniques

It can be seen from Figure 16 that detailed profiling takes huge amount of disk space (5.2 GBs) where as selective takes less than half of it (2GBs). Selective profiling gives user opportunity to filter the profile data and reduce the size of it. The compact profiling technique is most efficient in that it takes only 237.6MB, but as mentioned earlier, it requires a more sophisticated analysis tool.

4.3 Overhead of Profiling

This section shows the overhead of the profiling in terms of execution time for the SAT Solver. Figure 17 shows the overhead with respect to the number of variables in the input CNF formula. It can be seen that as the number of variables increase the time to profile that problem increases. This effect is due to the fact that we are capturing algorithmic variables, which ultimately depend on input formula size. The maximum slowdown was 2.32x for 360 variables in a CNF formula.

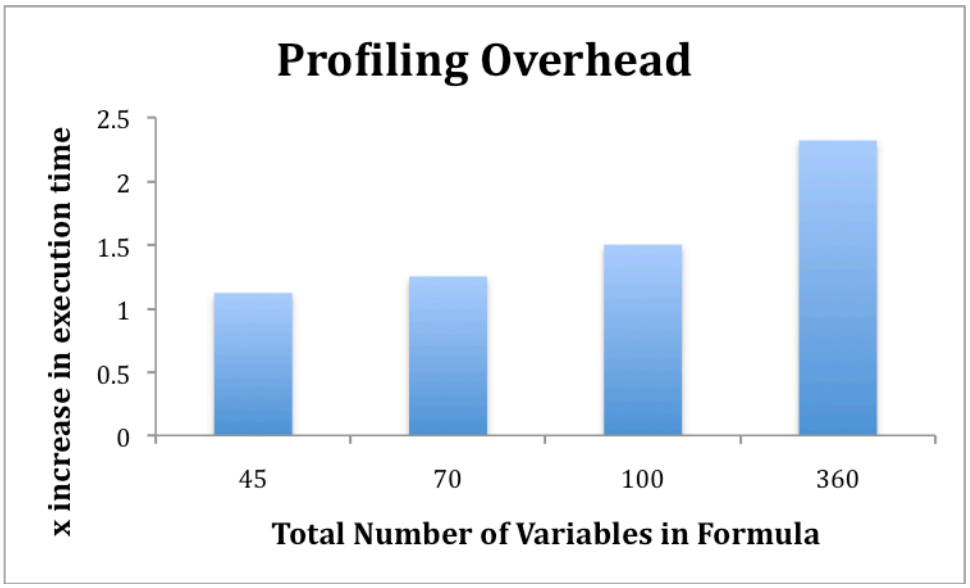

Figure 17: Profiling Overhead

CHAPTER 5

PERFORMANCE TUNING AND DEBUGGING OF SAT SOLVERS

In this chapter, we would like to discuss the effectiveness of our framework in detail over the traditional techniques used for choosing and tuning solvers by the users and for performance debugging by the designers.

5.1 Choosing and tuning a solver

Choosing and tuning the SAT solver to an application domain is difficult because each application domain presents different set of problems. Further, the problem becomes more difficult if a domain has different kinds of problems. For example, if domain contains combination of 3SAT and 7SAT problems then HaifaSAT is best for solving problems of type 3SAT, but does not perform well for problems of type 7SAT. This is shown in Table 1. So, the user can tweak the solver parameters so that it performs better with 7SAT. As we observed from our experiments, HaifaSAT does the first stable assignment for its variable in 2^{nd} iteration for 3SAT and 5SAT where as in 222^{nd} iteration for 7SAT. This explains that as the number of variables per clause is increased, HaifaSAT takes more and more preprocessing time. Thus, our framework can be used to find the bottleneck without actually getting into the details of the strategy implemented by the solver. It also shows that HaifaSAT should never be used for 7SAT problems problems. Using these different metrics and knowledge of the domain (such as the density of the SAT problem), users can choose and tweak different phases to yield best behaviors of the chosen solvers for the domain.

5.2 Performance Debugging

While the choice of a solver is a difficult problem, performance debugging of a solver by its designers can get even more difficult because every solver run presents huge data to be analyzed. In this section we show the use of our technique for debugging performance and we use results of our technique to improve WalkSAT, MiniSAT and zChaff's performance.

5.2.1 Performance Debugging of WalkSAT

Considering WalkSAT's performance, as shown in Table 8, the behavior shows that each variable is unstable for a very long interval (in terms of iterations) during the solution of a particular SAT instance. Figure 18 also shows that once the 20% of variables find their correct value for satisfying the formula, other variable values can be calculated quickly. Here we have used benchmarks from DIMACs [15], since it has a number of complex and practical problems.

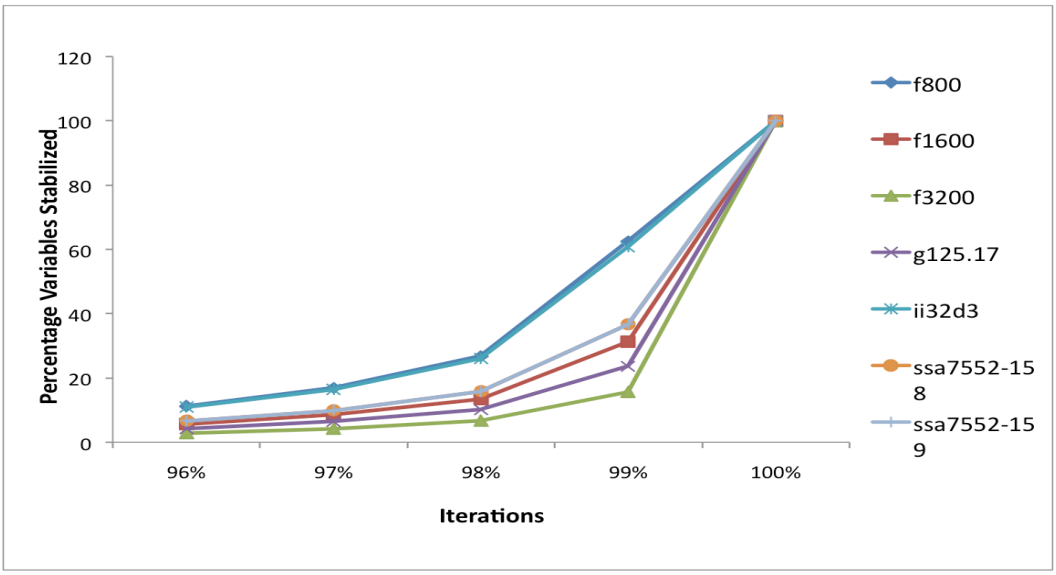

Figure 18: Pattern for Stabilization of variables for WalkSAT

This analysis shows that once we calculate the values for right variables correctly, the entire formula can be solved very easily. We use this analysis and try to reduce the cost of finding values of variables, which majorly contribute in satisfying the formula. Thus, the key to speeding up performance of WalkSAT is to be able to find the "seed" solutions quickly. For this purpose, we deploy a new programming framework developed by our group on n-version programming. The key idea behind n-version programming model is to launch as many and as diverse versions of the computations searching the solution space as possible. For WalkSAT, we try parallelizing the initial phase of finding right variable using as many and as diverse a set of randomizations as possible and we achieve considerable amount of speedup in finding the solution. The key hypothesis is that due to diverse randomizations some versions get lucky in getting closer to the initial solution thereby breaking the solution space. SAT solvers are not particularly amenable to "dividing the work" amongst parallel threads; such an approach based on n-versioning on the other hand, leverages the diversity attributable to randomization thereby probabilistically increasing the chances of getting to the right starting solution.

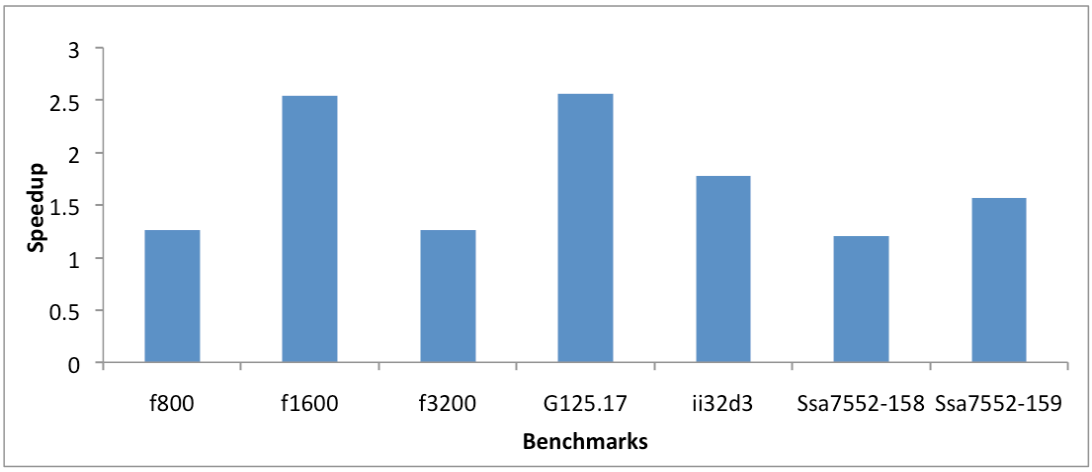

Figure 19: Speedup achieved using performance debugging for WalkSAT

We note that the effort required to "parallelize" and speedup WalkSat was minimal. The original algorithm and code remained basically unchanged minus a few calls to a runtime to manage the diversity and maintain correct algorithmic semantics. Figure 19 shows the results (in terms of speedup) we obtained by running various benchmarks with and without parallelization.

5.2.2 Performance Debugging of MiniSAT

In this section we demonstrate the performance debugging of MiniSAT using our analysis tool. For DPLL-based SAT Solvers, there are two algorithm steps which can vary a lot depending on the application domain (1) Deciding next branch (2) Conflict Analysis and resolution.

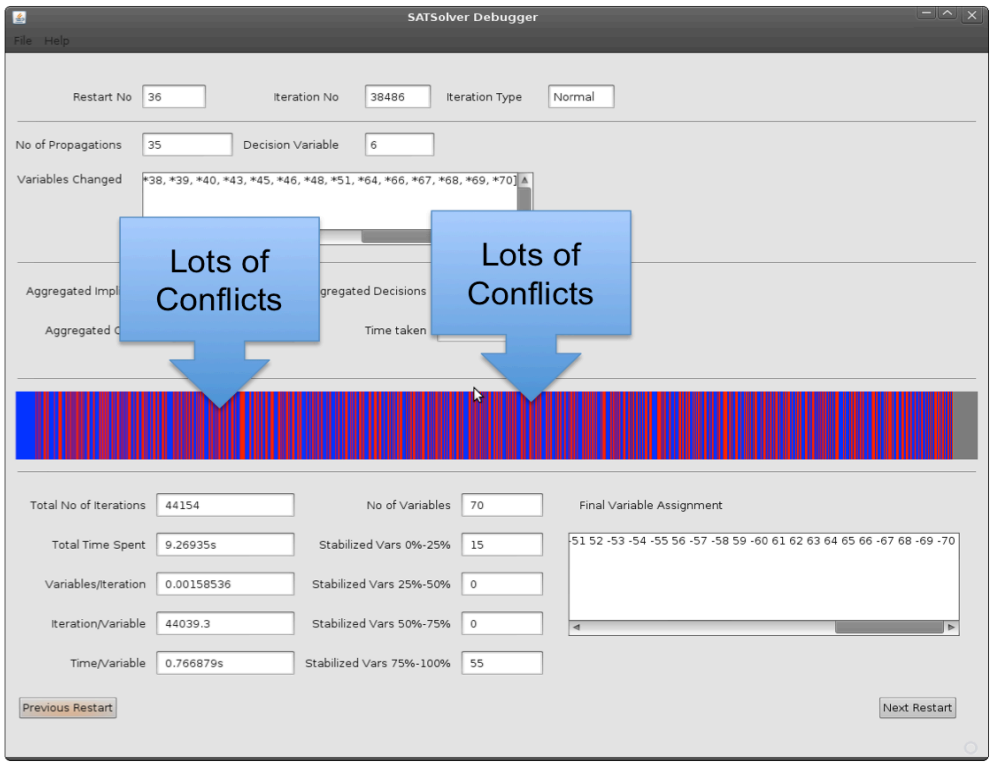

Figure 20: Performance Debugging of MiniSAT

Using our framework it can be found which one of them would be most effective for reducing the time to obtain solution. Figure 20 shows the run of MiniSAT for 3SAT problems. It can be seen that there are lot of high-density conflict areas and conflicts happen due to incorrect selection of decision variable. MiniSAT has two decision strategies 1) Random Decision 2) Activity-based decision. It uses the other one if one of them fails. If both of them are going to give the solution then the sequence in which they are applied decides which decision strategy should be used. So, using this debugging information, we change the sequence as to first use activity-based decision and then use random decision. Figure 21 shows the speedup we obtained by changing the decision variable selection strategy.

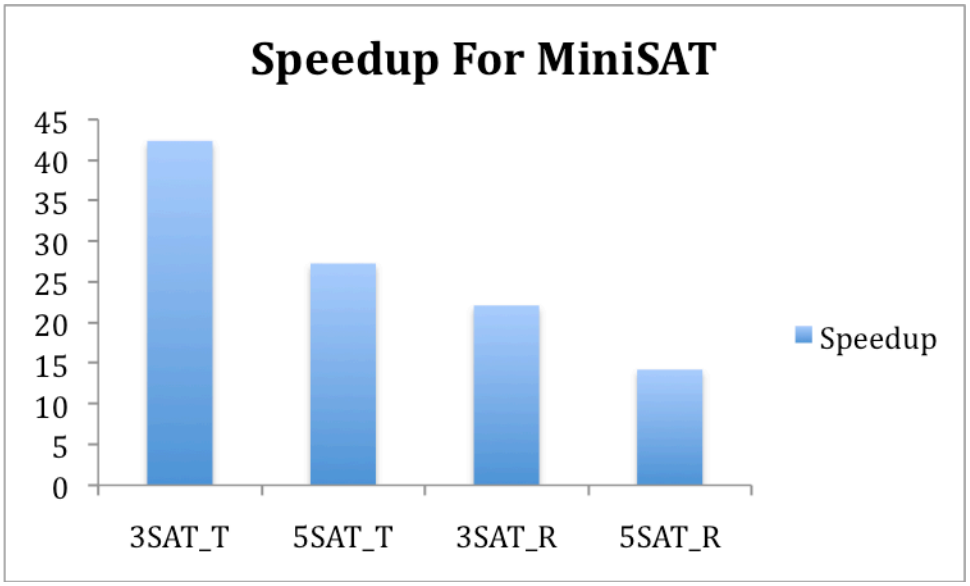

Figure 21 Speedup achieved using performance debugging for MiniSAT

With the change in variable selection strategy, it can be seen that for 3SAT problems the improvement is more than 45x for training problem and 22x for reference problem. For 5SAT problems improvement is more than 25x for training problems and more than 11x for reference problem. Same strategy does not work for 7SAT problems and time

required for obtaining the solution increases by 2x. This reiterates the fact that tuning of SAT Solver is important for every domain and particular problem type.

5.2.3 Performance Debugging of zChaff

We run our analysis tool on zChaff's profile data and it can be seen that zChaff's conflict analysis undo's lot of previous assignment which might not be necessary and slows it down.

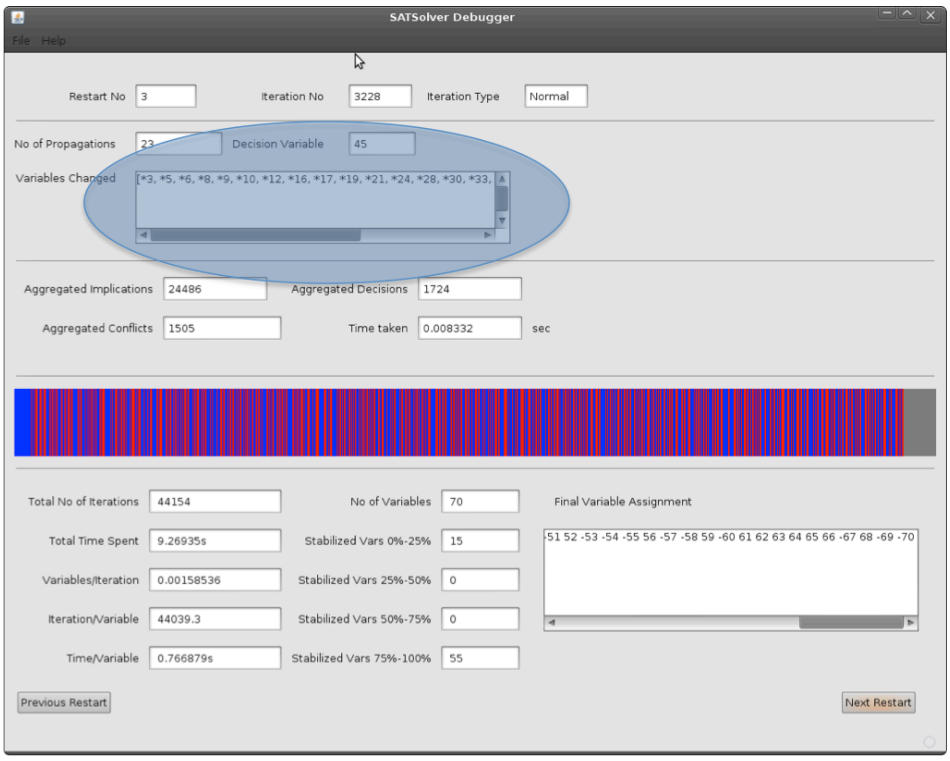

Figure 22 Performance Debugging of zChaff

There are different conflict analysis strategies implemented in zChaff, like *first unique implication point*, *first unique implication point resolved based*, *all unique implication point* and *decisions only* (more details on these strategies can be found in [18]). It is necessary to use appropriate strategy for an application domain to obtain solver results in minimum time. According to our debugging tool, it can be inferred that a lot of variables is undone for each conflict. This might help for some application domains but in this case it is causing solver to take too long to provide solution. The strategy, which undo's more

variables is all unique implication analysis and was used in the profiled solver. So, we changed the strategy for conflict analysis to first unique implication point analysis and Figure 23 shows the speed up that we got with this strategy.

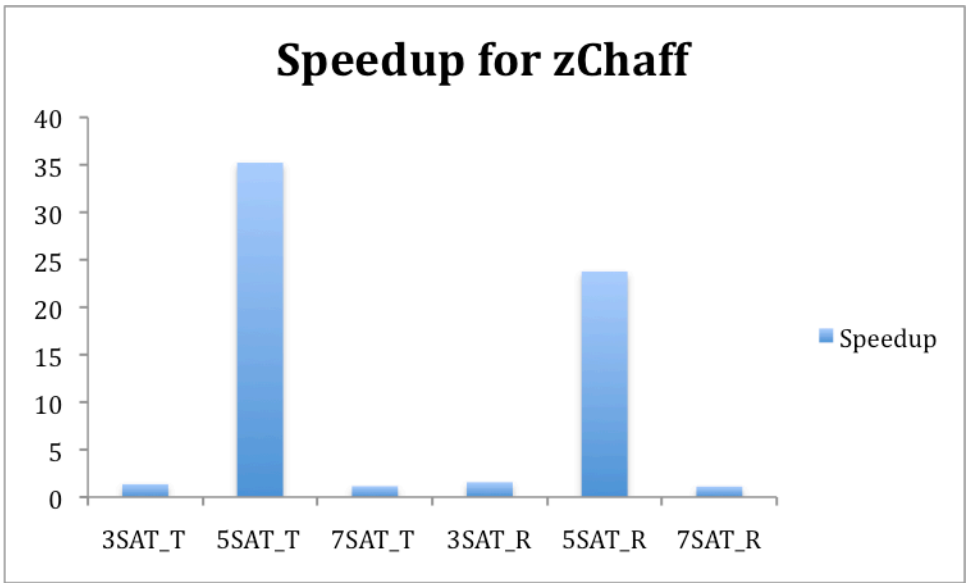

Figure 23 Speedup achieved using performance debugging for zChaff

It can be seen that for 5SAT problems the improved version gives 35x speedup for training problem and 23x for reference problem. For 3SAT and 7SAT problems it makes the solver run faster than the pervious version.

CHAPTER 6

GPU ARCHITECTURE BACKGROUND

In chapters 6 and 7, we show that our profiling framework is generic enough to capture the behavior of loops in general. We demonstrate the usefulness of our framework for general loops by providing hints to improve their performance using GPUs. We provide these hints by analyzing the branching behavior of an appliction and hence here, it is necessary to capture program state rather than algorithmic state.

In this chapter we provide a background of GPU architecture and programming model. We use NVIDIA's GPGPU programming model also referred to as Compute Unified Device Architecture (CUDA) [23] to demonstrate use of our profiling technique. Section 6.1 gives a brief overview of NVIDIA GPU architecture and section 6.2 gives an idea about the programming model.

6.1 GPU Architecture

Graphics Processing Units or GPUs are driven by the never-satisfied demand for real time, high definition 3D graphics and are evolved as highly parallel-multithreaded manycore processors [24].

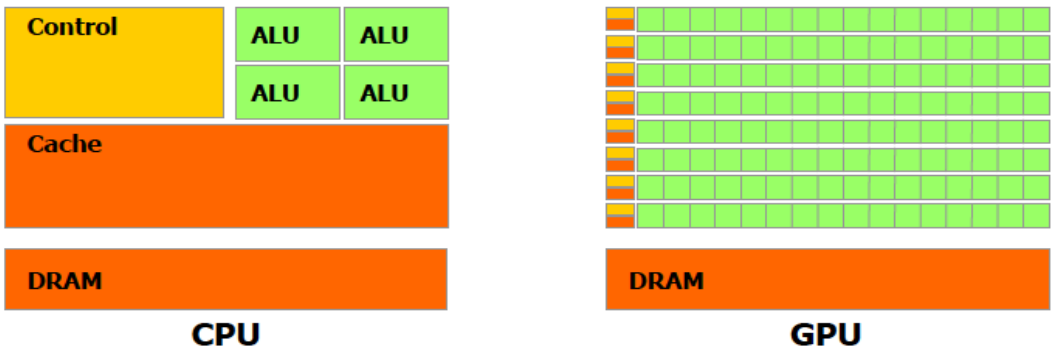

Figure 24: Transistor usage in CPU and GPU architecture (from [24])

They exhibit very high GFLOP/s performance than CPUs as it can be seen in Figure 25 The main reason to we get such a high performance is that GPUs devote more transistors to data processing rather than data caching and control flow as shown in Figure 24. A GPU consists of several Streaming Multiprocessors (SMs), which are connected to a common device memory. Modern GPUs have 8-32 SMs Each SM consists of about 8 Scalar Processors (SP). Since GPUs spend fewer transistors on control flow each SP shares a common instruction fetch unit with other SPs in an SM [25]. This forms a basis for GPU's Single Instruction Multiple Thread model. This execution model is exposed to the programmer in terms of group of threads called *warps* and they form basic unit of execution. A warp executes one common instruction at a time, so full efficiency is realized when all threads of the warp agree on their execution. Since there is only one instruction fetch unit per warp, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths are complete, threads are converged together.

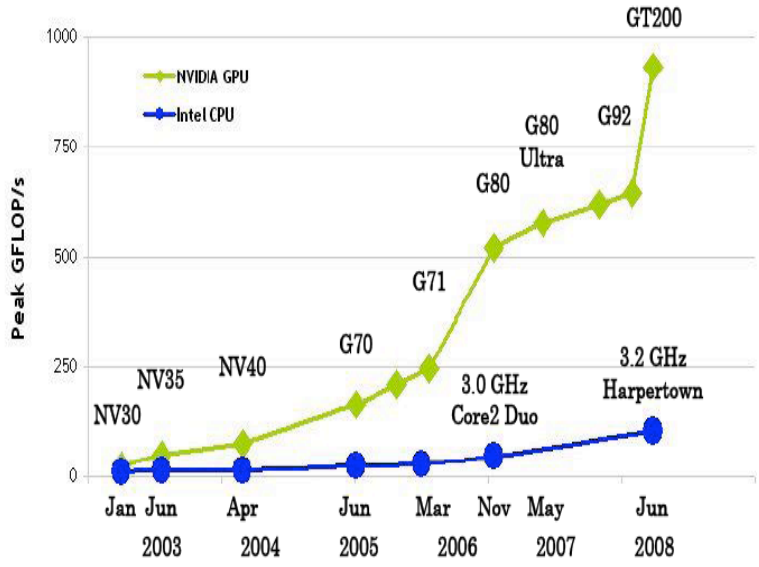

Figure 25: GPU-CPU performance comparison (from [24])

6.2 CUDA Programming Model

Programmer can write CUDA programs similar to that of C. NVIDIA extended the C language to include CUDA directives. Programmer can write functions to be executed on the GPU in CUDA C. These functions are called kernels. Kernels are executed in parallel by a number of CUDA threads unlike regular C functions, which are executed by a single thread.

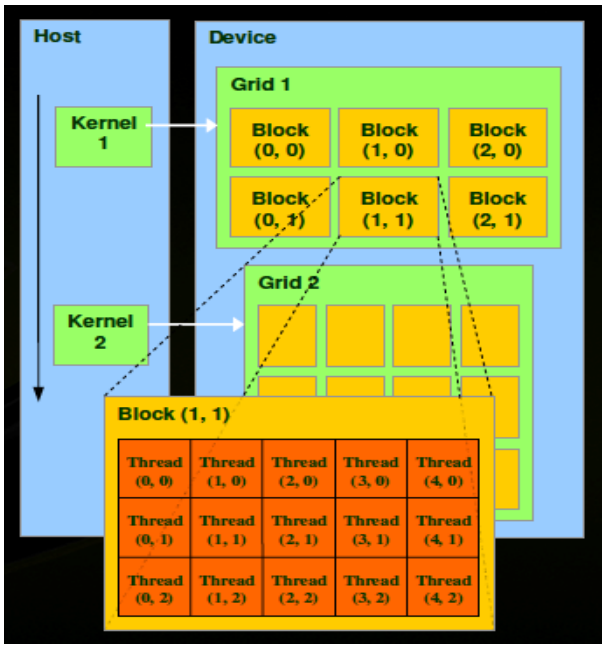

Figure 26: CUDA Programming Model

As shown in Figure 26 these kernels are organized in Grids and Blocks. A kernel is executed as a grid of thread blocks. A thread block is a batch of threads that can cooperate and synchronize with each other using shared memory and synchronization directives respectively. Two threads from different blocks cannot cooperate. When a SM is given one or more blocks to execute, it splits them into warps (in the Tesla architecture [26] warps are of size 32). The way blocks are split into warps is always same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing

thread 0. Any control flow instruction (if, switch, do, for, while) inside the kernel can affect the instruction throughput by causing threads of the same warp diverge. If this happens, the different executions paths are serialized, increasing total number of instructions executed for this block, degrading the instruction throughput. So, to get high instruction throughput and in turn peak performance, it is necessary to take care that threads in the same warp do not diverge. Our framework helps in reducing thread divergence while porting the branch heavy parallel loops to GPU.

CHAPTER 7

PROFILING GENERAL LOOPS

In this chapter, we show that the profiling data representation that is used to capture algorithm state of application can also be reused to capture program state of application. We profile parallel loops and capture the behavior of branches during each iteration. This profile data is used to find the branching behavior across the iterations. Later, this analysis is used find the loop's suitability for executing them on GPUs with respect to their branching behavior.

7.1 Profile Data Representation

We use the similar data representation that was used for SAT Solvers. Figure 27 shows the representation we use for profiling branches. For each loop iteration we profile each branch that was executed.

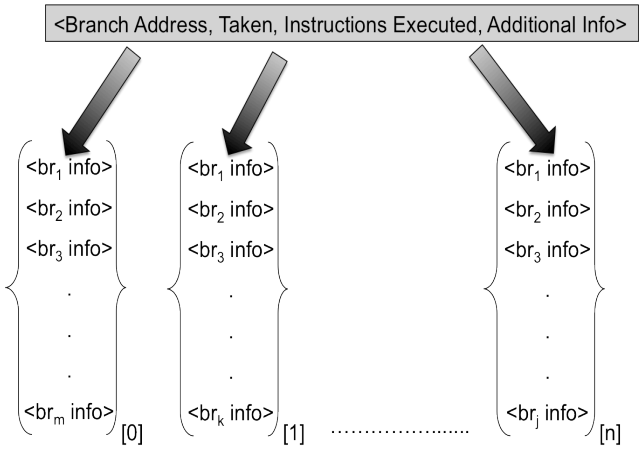

Figure 27: Profile Data Representation for general loops

For each branch we store its address, taken/not taken flag, instruction that are executed between the previous and this branch and some additional information such as flag to indicate whether it is backward branch. Since we capture information of dynamic branches, the number of branches profiled per iteration can differ. We also store some additional data to reduce the profiling data size like if the same branch occurs consecutively we store only one instance of it and add the repetition number. This makes our representation more compact as this takes care of high trip count inner loops with no branches inside them.

7.2 Framework

The basic framework remains the same; we capture the information about each iteration and store it on disk. The method we use for capturing the information (instrumentation and profiling) and further analyzing it differs. Following subsections explain the instrumentation and analysis components of the framework.

7.2.1 Instrumentation and Profiling

We use Pin dynamic instrumentation tool from Intel to perform the task of instrumentation and profiling. Figure 28 shows the basic architecture of Pin. Pin takes application's host ISA binary as input. It intercepts the execution of first instruction of the application. Its JIT compiler forms a trace out of application instructions and instruments by adding a call to instrumentation routine written in Pintool. The Dispatcher puts it into code cache and starts executing. The Pin takes care that after executing a trace

the control is given back to JIT compiler. The emulation unit is used to interpret the instructions that can not be directly executed e.g. system calls.

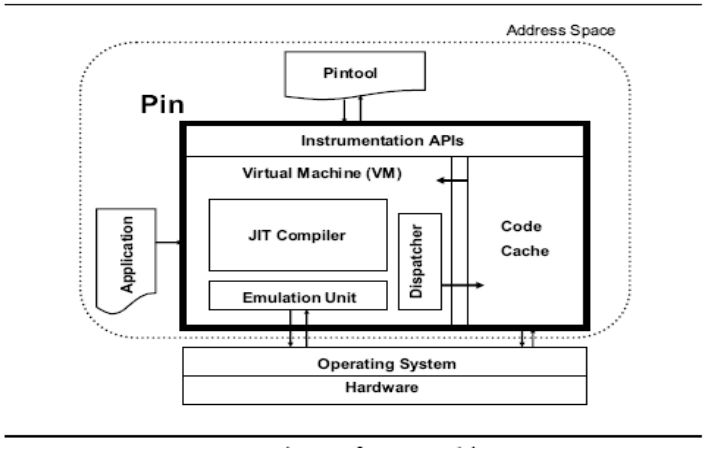

Figure 28: Pin Internal Architecture

The user of Pin can write its own customizable Pintool using the APIs exposed by basic Pin framework. We have written Pintool called "BranchAnalyzer" to instrument the branches and get the branching behavior of each iteration.

BranchAnalyzer:

Using BranchAnalyzer we instrument each function (routine) in the executable module. We do not instrument library functions because user has no control on the library function's branch behavior. Also, in many cases library function's implementation is different with respect to GPUs. We take input the source file and start and end line numbers of the loop to be profiled. Since the instrumentation is dependent on source file information it is necessary that debug information is present in the executable. Figure 29 shows the algorithm for instrumentation.

```
InstrumentRoutine()
  for each instruction I in Routine
    if(I in executable module)
      if(I is start of loop)
        add instrumentation before this instruction to start the
        profiling
      endif
      if (I is end of loop)
        add instrumentation before this instruction to stop the
        profiling
      endif
      if (I is branch instr with fallthrough)
        add instrumentation to gather information about this branch
      endif
      add instrumentation to count the instruction
    endif
  endfor
endofFunction
```

Figure 29: Instrumentation Algorithm

Using algorithm shown in Figure 28 we are able to profile branches across the functions; this makes the gathering of profile data inter-procedural and adds more accuracy to our analysis. After this instrumentation the application is executed and the profiling data is stored in a file. While storing the data in file we check for consecutive branches, which are same with respect to their address, condition and instructions executed and store them only once. This helps us in capturing inner loops with-no-branches efficiently. Since the information about which loop to be profiled is taken from user, our tool does not perform the task of automatically detecting the parallel loops.

This generated profile data is then fed to analysis tool to find the compatibility of the loop if it is directly (without any algorithmic modifications) ported to GPUs.

7.2.2 Analysis

We developed an analysis tool, which will take input branch information about each iteration of loop and performs branch divergence analysis. One more input to analysis tool is warp size, which will be used to know the number iterations to be combined for the analysis. Figure 30 shows algorithm we use to calculate branch divergence. The input to the algorithm is consecutive loop iterations that are equal in number to warp size, starting from 0, similar to that formed in GPUs.

```
processWarp(iteration_set, convergence_branch) // while invoking the
        // function with complete warp convergence_branch is ignored
  while (all branches in all iterations are not processed)
    if (current_branch is convergence_branch)
      return
    endif
    if(current_branch is taken or not taken by all iterations)
      current_branch = get_next_branch
      continue
    else
      Increment divergece_count
      divide the iterations into taken_itrs and not_taken_itrs
      if (branch is backward branch)
        call processWarp (taken_itr, next_branch)
      else
        call processWarp(taken_itrs, next_branch)
        call processWarp (not_taken_itrs, next_branch)
      endif
    endif
  endloop
endofFunction
```

Figure 30: Branch Divergence Analysis Algorithm

Using above algorithm we find the divergent branches and instructions that are serialized iteration group. Using the output of this analysis, thread divergence pattern can be found and it can be used for writing efficient CUDA kernels. Following subsections show application of this analysis on number of benchmarks.

7.3 Case Studies

Using above framework, we analyzed mpeg2enc application from mediabench [29], hotspot and streamcluster from Rodinia benchmark suite [28], and one individual application ray tracer. We ran our experiments on machine having Intel ® Xeon ® 2.67GHz and running Linux kernel version: 2.6.18.

7.3.1 mpeg2enc

MPEG is dominant standard for high-quality digital video transmission. The mpeg2enc application is responsible for encoding video frames into MPEG-2 [30] format and mpeg2dec application is used to achieve the reverse effect. Video frames are provided as input to mpeg2enc and it performs sequence of operations on them like motion estimation, predict, dct-type-estimation, transform, putpict, iquant, itransform and calcSNR. The video frames are combined into Group of pictures (GOP) for getting inter-frame compression. Three kinds of frames are involved in GOP are I, P and B. Basically, I frames are independent frames which do not depend on any other frames, P frames are encoded by using previous frames and B frames are encoded using previous and next pictures. (More details can be found in [30]). The GOP starts with I frame and can have combination of P and B frames that are decided by the designer. The GOP once set is repeated for entire video stream. The behavior of branches during each frame processing highly depends on the type of frame. So, using our tool we profile the mpeg2enc and find out the branching behavior of its frame-processing loop. Table 10 shows comparison of branch divergences between processing of all frames in GOP considered as one iteration and considering one frame processing per iteration. It can be seen that, if all frames in

GOP are processed as one iteration then there are very less branch divergences, compared to that of processing each frame of GOP as one iteration.

Table 10: Branch divergence for mpeg2enc

Configuration/Inputs	Percentage Branches Diverged		
	qos	dolbyaurora	dolbycity
All GOP frames processed in one iteration	5	13	11
One frame processed per iteration	55	67	63

So, if a kernel is formed to process all the frames in GOP together then, there would be less divergence among the iterations.

7.3.2 raytracer

In computer graphics, ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. This technique has higher degree of visual realism than scanline rendering methods, but at a greater computational cost. Since it is compute intensive application it is well suited for CUDA, but it also has number of branches interleaving the calculations, which can lead to its poor performance on GPUs. We used the raytracer application [32] developed by John Tsiombikas and Ian Mapleson to find out raytracer's branching behavior. The Figure 31 shows that raytracer has lot of branch divergences among its completely parallel loop. The Figure 32 shows the code that is serialized due to branch divergences.

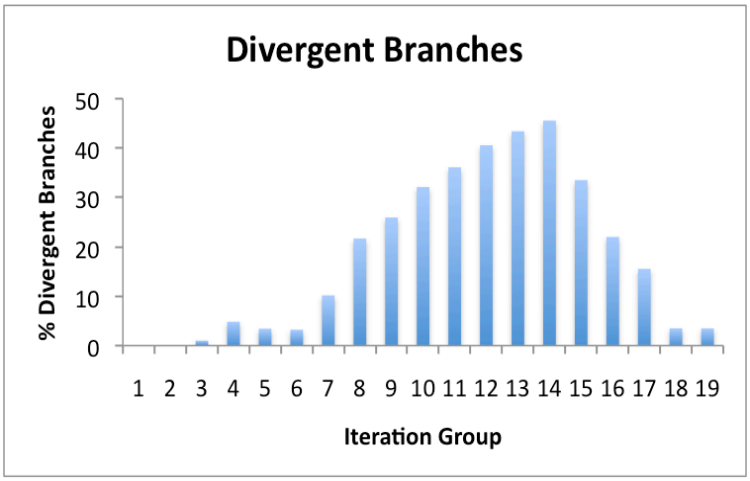

Figure 31: raytracer - Divergent Branches

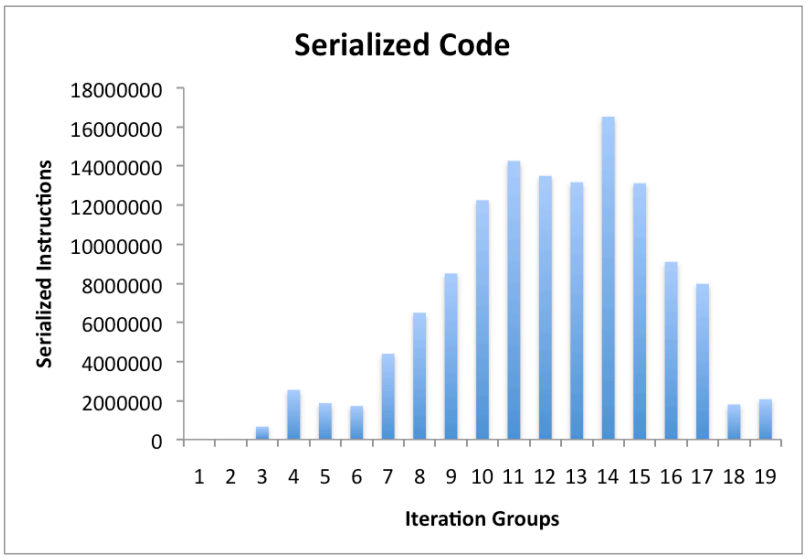

Figure 32: raytracer – Serialized Code

7.3.3 Hotspot

Hotspot [31] is widely used tool to estimate processor temperature based on an architectural floor plan and simulated power measurements. It is part of Rodinia benchmark suite. It performs the thermal simulation by iteratively solving a series of differential equation for block. We use its OpenMP version to know the parallel loop and gather the branching information of each iteration. Figure 33 shows the result of branch

divergence analysis and it can be seen that only initial few encounters the branch divergence. Rest of the iterations exhibit similar branch behavior. This happens because hotspot performs different calculations for corners of the floor. These calculations interfere with other parts calculation. So, if the corner calculations can be performed on CPU and other calculations on GPU, improved performance can be obtained.

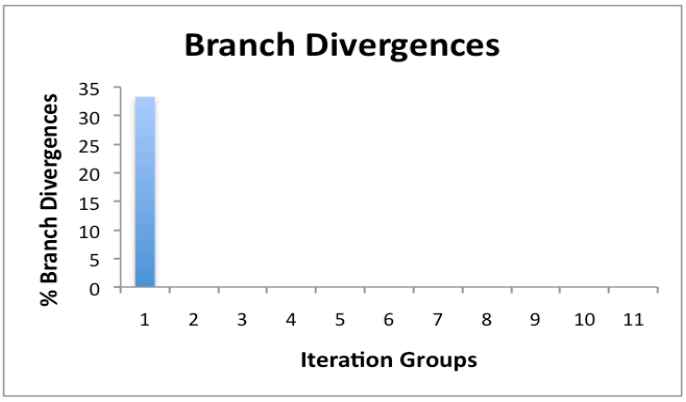

Figure 33: hotspot – Divergent Branches

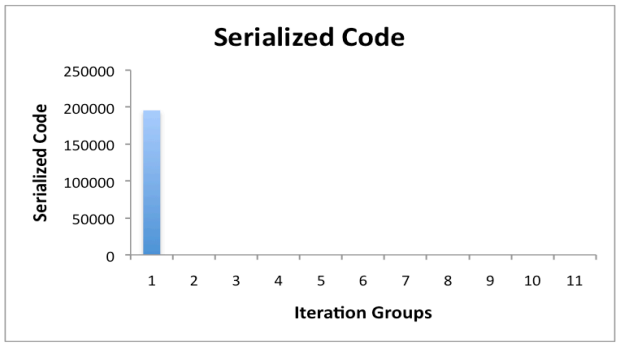

Figure 34: hotspot – Serialized Code

7.3.4 StreamCluster

Streamcluster finds the predetermined number of medians for a stream of input such that each point is assigned to its nearest cluster. The quality of clustering is measured by the

sum of squared distances (SSQ) metric. Streamcluster is also part of Rodinia benchmark suite. We use OpenMP version of Streamcluster to find the parallel loop and perform our analysis on it. Table 11 shows the result of our analysis, it can be seen that there is only single branch diverge across the 32 iterations considered together. So this loop is good to be ported as it is to GPU.

Table 11: Branches diverged for streamcluster

Iteration Group	Branches Diverged	Total Branches	Serialized Instructions
1	1	259	45
2	1	259	45
3	1	259	45
4	1	259	45
5	1	259	45
6	1	259	45
7	1	259	45
8	1	259	45
9	1	259	45
10	1	259	45
11	1	259	45
12	1	259	45
13	1	259	45
14	1	259	45
15	1	259	45
16	1	259	45

7.3.5 Hotspot implementation on GPU

```
for (r = 0; r < row; r++) {
      for (c = 0; c < col; c++) {
         /* Corner 1 */
         if ( (r == 0) && (c == 0) ) {
            delta = (step / Cap) * (power[0] +
                  (temp[1] - temp[0]) / Rx +
                  (temp[col] - temp[0]) / Ry +
                  (amb_temp - temp[0]) / Rz);
         }  /* Corner 2 */
         else if ((r == 0) && (c == col-1)) {
            delta = (step / Cap) * (power[c] +
                  (temp[c-1] - temp[c]) / Rx +
                  (temp[c+col] - temp[c]) / Ry +
                  (amb_temp - temp[c]) / Rz);
         }  /* Corner 3 */
         else if ((r == row-1) && (c == col-1)) {
            delta = (step / Cap) * (power[r*col+c] +
                  (temp[r*col+c-1] - temp[r*col+c]) / Rx +
                  (temp[(r-1)*col+c] - temp[r*col+c]) / Ry +
                  (amb_temp - temp[r*col+c]) / Rz);
         }  /* Corner 4 */
         else if ((r == row-1) && (c == 0)) {
            delta = (step / Cap) * (power[r*col] +
                  (temp[r*col+1] - temp[r*col]) / Rx +
                  (temp[(r-1)*col] - temp[r*col]) / Ry +
                  (amb_temp - temp[r*col]) / Rz);
         }  /* Edge 1   */
         else if (r == 0) {
            delta = (step / Cap) * (power[c] +
                  (temp[c+1] + temp[c-1] - 2.0*temp[c]) / Rx +
                  (temp[col+c] - temp[c]) / Ry +
                  (amb_temp - temp[c]) / Rz);
         }  /* Edge 2   */
         else if (c == col-1) {
            delta = (step / Cap) * (power[r*col+c] +
                  (temp[(r+1)*col+c] + temp[(r-1)*col+c] - 2.0*temp[r*col+c]) / Ry +
                  (temp[r*col+c-1] - temp[r*col+c]) / Rx +
                  (amb_temp - temp[r*col+c]) / Rz);
         }  /* Edge 3   */
         else if (r == row-1) {
            delta = (step / Cap) * (power[r*col+c] +
                  (temp[r*col+c+1] + temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx +
                  (temp[(r-1)*col+c] - temp[r*col+c]) / Ry +
                  (amb_temp - temp[r*col+c]) / Rz);
         }  /* Edge 4   */
         else if (c == 0) {
            delta = (step / Cap) * (power[r*col] +
                  (temp[(r+1)*col] + temp[(r-1)*col] - 2.0*temp[r*col]) / Ry +
                  (temp[r*col+1] - temp[r*col]) / Rx +
                  (amb_temp - temp[r*col]) / Rz);
         }  /* Inside the chip   */
         else {
               delta = (step / Cap) * (power[r*col+c] +
                  (temp[(r+1)*col+c] + temp[(r-1)*col+c] - 2.0*temp[r*col+c]) / Ry +
                  (temp[r*col+c+1] + temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx +
                      (amb_temp - temp[r*col+c]) / Rz);
         }
         /* Update Temperatures  */
         result[r*col+c] =temp[r*col+c]+ delta;
      }
}
```

Figure 35: hotspot – Parallel Loop Code

As it can be seen from Figure 35, Hotspot varies in its calculation for corners. So, if the corner calculations, that is for indexes (x=0, x=row-1, c=0 and c=col-1) are done on CPU then no branch divergences occur in CUDA implementation. Using the branch analysis and hints mentioned in 7.3.1.3, we implemented hotspot on GPUs. We performed all corner calculations on CPU and others are offloaded to GPU. This causes less branch convergence and improves hotspot by 6x with respect to its OpenMP counterpart.

CHAPTER 8

RELATED WORK

To the best of our knowledge, there is no similar work done in finding the iterative efficiency of the SAT solvers which can be used in performance analysis and tuning. SAT solvers have always been treated as black-boxes and their performance tuning and debugging mainly remains an ad-hoc art. Some attempts have been made to give valuable visualization aid in this domain. Cameron Brien and Sharad Malik at Princeton University have developed a visualization tool called TIGERDISP [8] for visualizing runtime behavior of DPLL based solvers. Difference between our framework and TIGERDISP is that our framework is not restricted to DPLL based solvers and can be applied to almost every type of solver without any knowledge of its internals though most of the solvers used in this study are DPLL. Also, TIGERDISP shows the solver behavior in terms of graph showing relationships between variables and clauses. But as the problem size increases this graph can become complex and difficult to comprehend. In contrast with this, our analysis tool is based on summary information and its complexity is independent of problem size. In addition, our framework gives users enough freedom to build their own performance model according to the application domain by combining the basic metrics provided. The performance model can be built on set of metrics that are supported and then can be used to choose the solver.

With respect to profiling general loops there is work on translating OpenMP programs to CUDA by Lee et al. [22] and C to CUDA by Baskaran et al. [34] In both of these compiler frameworks they do not use profile data to assist their translation process.

There is also profiling work on CUDA kernels by Andrew et al. [33], but they use the CUDA kernel and their runtime environment to capture the application behavior, where as in our work we directly deal with x86 application binary and use manual detection of parallel loops. There is also work on collecting the control flow information using profiling by James Larus [35], which tries to compactly represent the control flow information of entire program execution. Our work differs from this in a way that we are trying to gather data about parallel loops and in that specifically about branches, further we use this data and analyze iterations and branch behavior collectively.

CHAPTER 9

CONCLUSION

In this thesis, we describe a framework, which can be used to analyze performance of an application using either Algorithmic State or Program State.

We capture algorithmic state by adapting our framework for SAT Solvers. It provides a generic and easy to use instrumentation system, which can be easily applied to any type of solver by simple insertion of generic APIs. Further, the metric calculation module is generic in that user can define its own performance metric and models. An analysis tool is developed to analyze the profile data and performance debug and tune the solver for given domain. Using the performance debugging provided by this framework, we show that MiniSAT and zChaff solvers can be effectively tuned to problem domain. Also, performance debugging data generated by our tool, in conjunction with a new parallel programming model based on n-versioning, we show we can effectively speed up solvers such as WalkSAT with minimal effort. In addition, this framework allows one to quantitatively analyze sub-steps in SAT solvers and one could therefore envision composing an efficient solver using those sub-steps.

We capture Program State by adapting our framework to parallelizable loops and capturing their branch behavior for each iteration. We show that this information can be useful for porting parallel to GPUs.

CHAPTER 10

FUTURE WORK

In future, we would like to support more metrics to handle additional aspects of SAT solver, so that efficiency can be explained at finer granularity. In particular, we would like to support data structure based metrics that would help to debug them in conjunction to the purely iterative behavior we are currently handling. We would also like to extend this approach to be useful for other iterative convergence based computation problems, such as physics simulations.

APPENDIX A

LIST OF ABBREVIATIONS

SAT	Boolean Satisfiability Problem
GPU	Graphics Processing Unit
CNF	Conjunctive Normal Form
SM	Streaming Multiprocessor
SP	Scalar Processor
GOP	Group of Pictures

REFERENCES

[1] M. Velev, and R. Bryant, Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors, Proc. of the Design Automation Conference, 2001.

[2] R.K. Brayton et. al. Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, -1984.

[3] J.Burch, V.Singhal. Tight Integration of Combinational Verification Methods Proceedings of International. Conf. on Computer-Aided Design.-1998.

[4] E. Goldberg, M.Prasad. Using Sat for combinational equivalence checking Proceedings of Design, Automation and Test in Europe Conference. –2001. - P.114-121.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs Proceedings of Design Automation Conference, DAC'99. -1999.

[6] Wikipedia (Accessed on April 1st 2009)
http://en.wikipedia.org/wiki/Boolean_satisfiability_problem

[7] A. Braunstein, M. Mezard, R. Zecchina. Survay Propagation: an algorithm for satisfiability. Random Structures & Algorithms Volume 27, Issue 2 (September 2005) Pages: 201 – 226, Year of Publication: 2005, ISSN: 1042-9832

[8] Cameron Brien, Sharad Malik: Understanding the Dynamic Behaviour of Modern DPLL SAT Solvers through Visual Analysis, Formal Methods in Computer Aided Design, 2006. FMCAD apos;06

[9] Niklas Een, Niklas Sörensson: MiniSat — A SAT Solver with Conflict-Clause Minimization, poster for SAT 2005.

[10] Zhaohui Fu, Yogesh Marhajan, Sharad Malik: ZChaff2004: An Efficient SAT Solver, Theory and Applications of Satisfiability Testing, 7th Intl' Conference, SAT 2004.

[11] Wikipedia (Accessed on April 1st 2009) http://en.wikipedia.org/wiki/DPLL_algorithm

[12] Wikipedia (Accessed on April 1st 2009) http://en.wikipedia.org/wiki/WalkSAT

[13] Roman Gershman, HaifaSat: a new robust SAT solver, S. Ur, E. Bin, and Y. Wolfsthal (Eds.): Haifa Verification Conf. 2005, LNCS 3875, pp. 76–89, 2006. Copyright. Springer-Verlag Berlin Heidelberg 2006

[14] http://en.wikipedia.org/wiki/CPLEX Accessed on 26th April 2010

[15] http://dimacs.rutgers.edu/dimacs2.html Accessed on 26th April 2010

[16] Niklas S¨orensson, Niklas Een, MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization

[17] Zhaohui Fu, Yogesh Marhajan, Sharad Malik, zChaff SAT Solver

[18] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, Sharad Malik, Efficient Conflict Driven Learning in a Boolean Satisfiability Solver

[19] Thesis by Lintao Zhang, Searching for truth: Techniques for satisfiability of Boolean formulas, Princeton University

[20] Lintao Zhang, SAT Course slides, IT-U of Copenhagen, November 2003

[21] Zhuo Huang, Prabhat Mishra, SAT-based Combinational Equivalence Checking, CSIE Technical Report #05-007, University of Florida, Gainesville, FL, August 2005

[22] Seyong Lee, Seung-Jai Min and Rudolf Eigenmann, OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization, Annual Symposium on Principals and Practices of Parallel Programming 2009

[23] NVIDIA CUDA [online]. Available: http://developer.nvidia.com/object/cuda_home.html Accessed on 26th April 2010.

[24] NVIDIA_CUDA_ProgrammingGuide 3.0, NVIDIA

[25] Effect of Instruction Fetch and Memory Scheduling on GPU Performance, Nagesh B. Lakshminarayana, Hyesoon Kim, Workshop on Language, Compiler, and Architecture Support for GPGPU, in conjunction with HPCA/PPoPP 2010, 2010.

[26] E. Lindholm, J. Nickolls, S.Oberman and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, 28(2):39–55, March-April 2008.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

[28] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing." In Proceedings of the

IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct. 2009.

[29] Chunho Lee , Miodrag Potkonjak , William H. Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems, Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture, p.330-335, December 01-03, 1997, Research Triangle Park, North Carolina, United States

[30] MPEG-2 White Paper – Pinnacle, Feb 2000.

[31] W. Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, M. R. Stan, Hotspot: A compact thermal modeling methodology for early-stage VLSI design., IEEE Transactions on VLSI Systems 14 (5) (2006) 501 – 513

[32] http://www.futuretech.blinkenlights.nl/c-ray.html Accessed on 26th April 2010

[33] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili, A Characterization and Analysis of PTX Kernels, IEEE International Symposium on Workload Characterization 2009

[34] M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs ," International Conference on Compiler Construction (CC), Paphos, Cyprus, March 2010.

[35] James R. Larus, Whole Program Paths, Proceedings of the SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99), May 1999, Atlanta Georgia.